Hardness of Games and Graph Sampling

Bhagirathi Nayak

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

Department of Computer Science and Engineering

June 2018

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

Bhagirathi Nayak

(Signature)

(Bhagirathi Nayak)

CS16MTECH11004

(Roll No.)

Approval Sheet

This Thesis entitled Hardness of Games and Graph Sampling by Bhagirathi Nayak is approved for the degree of Master of Technology from IIT Hyderabad

M.V. Panduranga Rao

(M V P Rao) Examiner
Dept. of Computer Science and Engg.
IITH

(———-) Examiner
Dept. of Computer Science and Engg.
IITH

N R Aravind

(N.R. Aravind) Adviser
Dept. of Computer Science and Engg.
IITH

(MARIA FRANCES) Co-Adviser Examiner
Dept. of Computer Science and Engg.
IITH

Subrahmanyam Kalyanasundaram

(Subrahmanyam Kalyanasundaram) Chairman
Dept. of Computer Science and Engg.
IITH

Acknowledgements

Firstly, I would like to express my sincere gratitude to my thesis advisor Dr. N.R. Aravind for the continuous support in my thesis work and related research, for his patience, motivation, and immense knowledge in theory. His guidance has helped me to think like a researcher, he has also reviewed the presentations and this thesis work which resulted in writing a better content for the readers.

I would also like to thank the professors for their valuable comments and suggestions during the presentations. It has helped me to think in new directions and improve my research work.

Finally, I would express my deep gratitude to my parents for providing me tremendous support throughout my research work. This achievement wouldn't have been possible without them.

Dedication

.

Abstract

The work presented in this document is divided into two parts. The first part presents the hardness of games and the second part presents Graph sampling. Non-deterministic constraint logic[1] is used to prove the hardness of games. The games which are considered in this work is Reversi (2 player bounded game), Peg Solitaire (single player bounded game), Badland (single player bounded game). It also contains a theoretical study of peg solitaire on special graph classes. Reversi is proved to be PSPACE-Complete using Bounded 2CL, Peg Solitaire is proved to be NP-Complete using Bounded NCL. Badland is proved to be NP-Complete by a reduction from 3-SAT. The objective of study of peg solitaire of special graph classes is to find the maximum number of marbles we can remove from a fully filled board, if the player is given the privilege to remove a marble from any cell initially, then following the rules after the initial move.

The second part of the work is dedicated to graph sampling. Given a graph G, we try to sample a representative subgraph G_s which is similar to the original graph G. The properties that are being studied are Degree Distribution, Clustering Coefficient, Average Shortest Path Length, Largest Connected Component Size. To measure the similarity between the original graph and sample we use the metrics Kolmogorov - Smirnov test and Kullback - Leibler divergence test. Tightly Induced Edge Sampling performs well on general graphs but it's performance decreases when the graph is a tree. Overall TIBFS and KARGER produces a sample which closely matches the distribution of original graphs.

Contents

Chapter 1

Introduction

People all over the world entertain themselves by playing games. Over the years researchers have found interesting connections among games, puzzles and computations. As computer scientists, we find that games and puzzles serve as powerful models of computation, quite different from the usual models of automata and circuits, offering a new way of thinking about computation.

1.1 Preliminaries

1.1.1 Game

A game is an activity or sport usually involving skill, knowledge, or chance, in which you follow fixed rules and try to win against an opponent or to solve a puzzle.

- Board Games
  - chess
  - checkers
  - Go
- Card Games
  - Poker
  - Bridge
- One Player Games
  - Rush Hour
  - Peg Solitaire
  - Sliding Puzzles

1.1.2 Bounded Games

The games in which the number of moves that can be made by a player is polynomially bounded are bounded games. For example, in peg solitaire, in a single move a player removes a marble from the board. The game ends when there is no move or there is only one marble left on the board.

1.1.3 Unbounded Games

The games is which there is no bound on the number of moves that can be played by a player, typically the moves are reversible. Examples include Chess, Checkers and Go.

1.1.4 One Player Games

A one-player game is basically a puzzle where one player makes a series of moves to accomplish a goal.

1.1.5 Two Player Games

In a two-player game, players alternate making moves, each trying to achieve some particular objective.

1.1.6 Team Games

In team game, players from team make moves, each team trying to achieve some particular objective.

1.2 Constraint Logic Formalism

1.2.1 Constraint Graph

1.2.1.1 Definition

It is a weighted directed graph which consists of vertices and edges. Edges are of two types which are as follows:

- Red Colored Edge having weight of 1
- Blue Colored Edge having weight of 2

1.2.1.2 Constraint of the graph

Each vertex has a non-negative minimum inflow. A legal configuration of a constraint graph has an inflow of at least the minimum inflow at each vertex. A legal move on a constraint graph is the reversal of a single edges orientation, resulting in a legal configuration. In this report we consider the minimum inflow to be 2.

1.2.1.3 General Problem

Can a given edge be reversed which still satisfies the constraint?

1.3 Basic vertices for One player games

1.3.1 Bounded Games

1.3.1.1 OR

A vertex with incident edge weights of 2, 2, and 2 behaves as a logical OR. A given edge may be directed outward if and only if at least one of the other two edges is directed inward. We will call such a vertex an OR vertex. Figure 1.1 beside represents OR vertex.

Figure 1.1: OR vertex

1.3.1.2 AND

The blue edge having weight of 2 may be directed outward if and only if both red edges having weight of 1 each are directed inward. Otherwise, the minimum inflow constraint of 2 would not be met. We will call such a vertex an AND vertex. Figure 1.2 beside represents AND vertex.

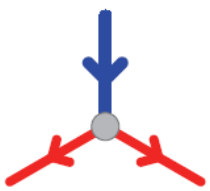


Figure 1.2: AND vertex

1.3.1.3 FANOUT

Both the red edges having weight of 1 each may be directed outward if and only if the blue edge having weight of 2 is directed inward. Otherwise, the minimum inflow constraint of 2 would not be met. We will call such a vertex an FANOUT vertex. It seems as if the weight of 2 from blue edge is fanning out through two red edges. Figure 1.3 beside represents FANOUT vertex.

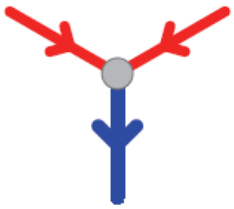


Figure 1.3: FANOUT vertex

1.3.1.4 CHOICE

A certain edge is pointed inward then only one of the 2 remaining edges can point outward. Otherwise, the minimum inflow constraint of 2 would not be met. It seems as if we came by an edge and choose one of the 2 remaining edges to move forward. Figure 1.4 beside represents CHOICE vertex.

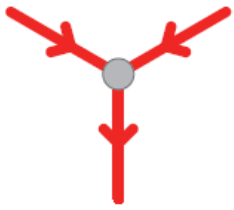


Figure 1.4: CHOICE vertex

1.3.1.5 GENERAL PROBLEM

Hardness: BOUNDED NCL IS NP-COMPLETE.[2]

1.4 Basic vertices for Two Player Games

1.4.1 Bounded Games

1.4.1.1 OR

A vertex with incident edge weights of 2, 2, and 2 behaves as a logical OR. A given edge may be directed outward if and only if at least one of the other two edges is directed inward.We will call such a vertex an OR vertex. Figure 1.6 beside represents OR vertex.

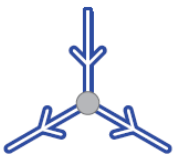


Figure 1.6: OR vertex

1.4.1.2 AND

The blue edge having weight of 2 may be directed outward if and only if both red edges having weight of 1 each are directed inward. Otherwise, the minimum inflow constraint of 2 would not be met. We will call such a vertex an AND vertex Figure 1.7 beside represents AND vertex.

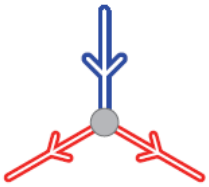


Figure 1.7: AND vertex

1.4.1.3 FANOUT

Both the red edges having weight of 1 each may be directed outward if and only if the blue edge having weight of 2 is directed inward. Otherwise, the minimum inflow constraint of 2 would not be met. We will call such a vertex an FANOUT vertex. It seems

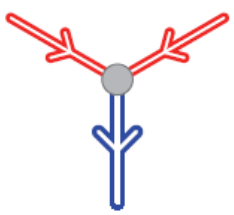


Figure 1.8: FANOUT vertex

as if the weight of 2 from blue edge is fanning out through two red edges. Figure 1.8 beside represents FANOUT vertex.

1.4.1.4 CHOICE

A certain edge is pointed inward then only one of the 2 remaining edges can point outward. Otherwise, the minimum inflow constraint of 2 would not be met. It seems as if we came by an edge and choose one of the 2 remaining edges to move forward. Figure 1.9 beside represents CHOICE vertex.

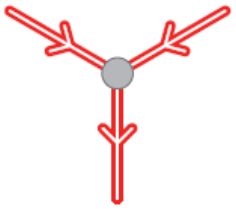


Figure 1.9: CHOICE vertex

1.4.1.5 VARIABLE

In variable gadget there are two edges one colored white and the other colored black. If The white player chooses this variable first, then it reverses the white edge and now black player can't choose this variable and reverse this edge. Similarly, if black player chooses first then only the black player can reverse the black edge. Figure 1.10 beside represents VARIABLE vertex.

Figure 1.10: VARIABLE vertex

1.4.1.6 GENERAL PROBLEM

TWO-PLAYER CONSTRAINT LOGIC (2CL)

Instance: Constraint graph G, partition of the edges of G into sets B and W, and edges $e_B \in B$, $e_W \in W$.

Question: Does White have a forced win in the following game? Players White and Black alternately make moves on G. White (Black) may only reverse edges in W (B). White (Black) wins if he ever reverses e_W (e_B).

Hardness: BOUNDED 2CL IS PSPACE-COMPLETE.[2]

Chapter 2

Studied Games

2.1 Reversi

2.1.1 Introduction

The hardness of generalized versions of games have been studied in past. The problem of determining whether a particular player can win is found to be EXP-TIME complete for checkers, chess, etc. It is PSPACE-complete for games like amazons, konane. Reversi has already been proved to be PSPACE-complete by Shigeki Iwata and Takumi Kasai[3] by a reduction from Generalized geography played on a bipartite graph having maximum degree of 3. In this section we will be proving Reversi is PSPACE-complete using Bounded 2CL which is much simpler to comprehend.

Reversi is a two-player strategy board game played on a n x n board. Figure 2.1 shows the initial board position. The board contains disks which are low on one side and dark on another side. Out of the two players, one of the player is assigned the low(dark)side and the other player with dark(low) side. The game progresses by players making moves alternately. The game continues till none of the player has a move to make. The objective of the game is to have majority of the disks turned to display one's color. A move corresponds to flipping at least one disk of opponent's color which are in a straight line (horizontally, vertically, diagonally) between the already placed player's disk and the disk going to be placed in this current move.

2.1.2 Basic Gadgets

The number of moves on a n x n board is polynomially bounded by n. Hence Reversi is a bounded two-player game. we will reduce Erik's bounded 2CL to reversi. The bounded 2CL consists of 5 gadgets which are as follows:

1. OR
2. AND, SPLIT
3. VARIABLE
4. CHOICE

and few utility gadgets (wire, turns, shifter). In the following sections we will denote player A as white player and player B as black player. White player will win if the number of white(low) side of disk outnumbers the black(dark) side of the disk and vice versa. The cells marked with alphabets are empty and any of the player can tap on it and make a move if it is valid.

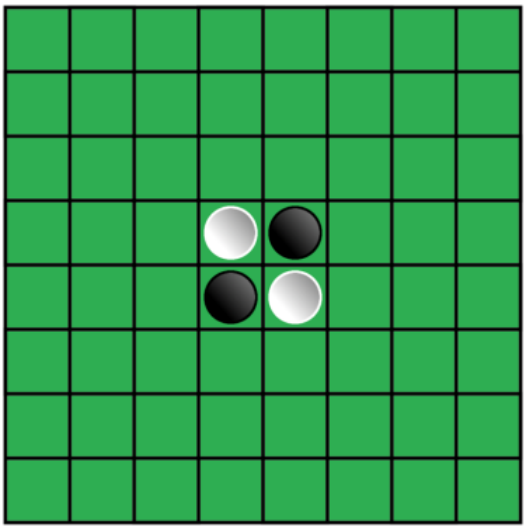


Figure 2.1: Initial board position

2.1.2.1 UTILITY

Figure 2.2 shows the utility Gadget. It consists of basic wiring. The white player can tap on cell E then it will tap on cell F to reverse the disks to white. To perform a shift, the white player can tap on A, then on cell B and finally cell D. To take a turn it can tap on cell A then B and finally on cell C. The gadgets are helpful in connecting the gadgets discussed below.

2.1.2.2 OR

Lemma 2.1.1. In an OR gadget white player can put a marble with white facing up on cell marked C if either it can put a marble with white facing up on A or on cell C.

Proof. Figure 2.3 shows the OR Gadget. If white player can tap on A by making a move. Then it can tap on B in his next move which will result in reversing the disk with black facing up between them to white. Now in his next move white player can tap on C and reverse the disk with black facing up between them to white. After these three moves the cell C contains disk with white facing up. Similarly, if the white player can tap on B by making a move, it will end up having disk with white facing up on cell C. There is no way the black player can resist in between the three moves the white player is going to make if we can connect gadgets wisely. For instance, it is not wise to tap A from below and keep a vacant cell to the left of A. The black player can make a move to left of A and the whole purpose of gadget will be lost. □

2.1.2.3 AND,SPLIT

It's a multi-purpose gadget which serves as both AND and SPLIT gadget. The flow is from INPUT 1 to OUTPUT 1 and INPUT 2 to OUTPUT 2. The flow corresponds to turning all the marbles white facing up from B to D and A to C.

Lemma 2.1.2. In an AND gadget white player can put a marble with white facing up on cell marked C iff it can put a marble with white facing up on A and on cell B.

Proof. Figure 2.4 shows the AND Gadget. To use it as an AND gadget we use INPUT 1, INPUT 2, OUTPUT 2. We feed the OUTPUT 1 to garbage or considered lost. We have to maintain the following flow order: INPUT

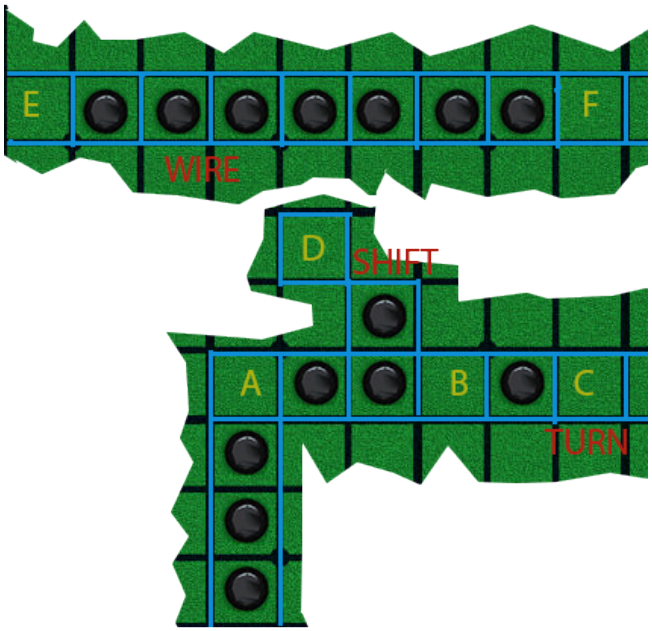


Figure 2.2: WIRE,TURNS,SHIFT gadget

1 to OUTPUT 1(to garbage) then INPUT 2 to OUTPUT 2. The reverse order will not serve the purpose of this gadget. If the white player activates INPUT 2 first, then eventually it will tap on cell H. Then the black player can tap on cell G and reverse the white facing on cell H. Hence the white player can't make move now. But if INPUT 1 activates first then above scenario is not possible. The sequence of moves are as follows: [B, I, J, K, L, D], [A, F, H, C]. Hence the white player need to tap both on cell A and B to finally tap C. □

Lemma 2.1.3. In a SPLIT gadget white player can put a marble with white facing up on cell marked C or D or both if it can put a white marble on cell B.

Proof. Figure 2.4 shows the SPLIT Gadget. To use it as a SPLIT gadget a free disk with white facing up is provided on INPUT 2. So if the white player can tap on cell B then it will finally tap cell D. As at INPUT 2 a free disk with white facing up is available the white player can freely use it to tap cell C as explained in the AND gadget above. □

2.1.2.4 VARIABLE

Lemma 2.1.4. In a VARIABLE gadget the player who choses first is able to reverse disks on its path while the other player has no valid move to make in that gadget.

Proof. Figure 2.5 shows the VARIABLE Gadget. If white player chooses first to make a move on this gadget then it taps on A, then B, then finally out of the gadget while the black player has no moves to make in this gadget. Similarly, if black player chooses first then white player is left with no moves to make in this gadget. In the context of constraint logic graph this is the vertex where we set variable to true or false. The variable is set to true if white player chooses else false. □

2.1.2.5 CHOICE

Lemma 2.1.5. In a CHOICE gadget if white player initially taps on cell A then it finally taps either B or C but not both.

Proof. Figure 2.6 shows the CHOICE Gadget. The white player taps on cell A. It finally taps on cell O after putting two white stones, one at cell D and another at cell J. The two white stones help in preventing the white

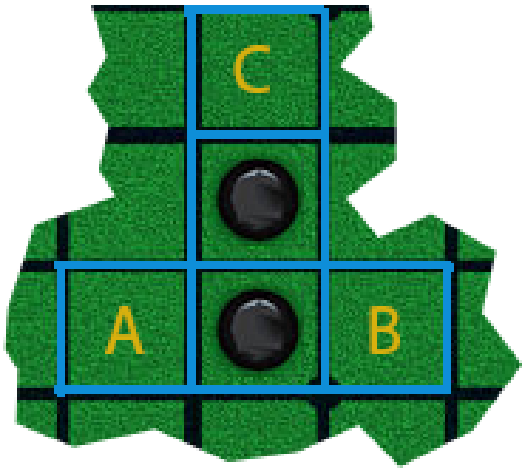


Figure 2.3: OR gadget

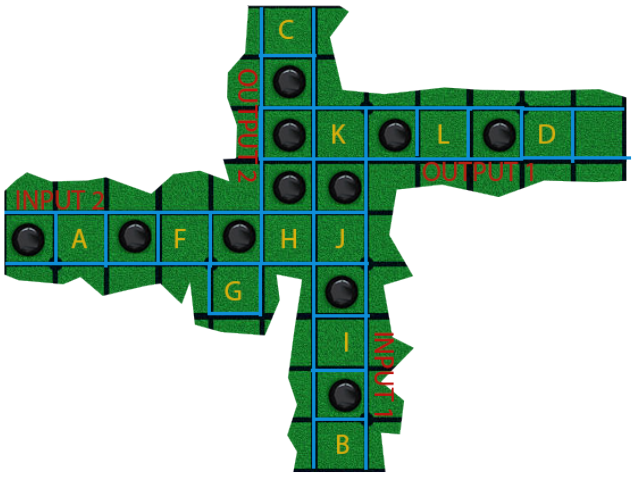


Figure 2.4: AND,SPLIT gadget

player from tapping both cell B and cell C. If the white player taps on cell R then there is no valid move to tap B. On the other hand, If the white player taps on cell Q then there is no valid move to tap C. So the white player can either tap cell B or cell C but not both. □

2.1.2.6 VICTORY

We will have an AND gadget whose output may be activated only if the white target edge in the 2CL game can be reversed, we need to arrange for White to win if he can activate this AND. We feed this output signal into a victory gadget, shown in Figure 2.7. we will arrange the board in such a way that at the end if black player gets X disks with black facing up. If white is able to tap on cell E he should get more than X disks with white facing up and hence win the game else he will lose.

Theorem 1. Given a n x n Reversi Board it is PSPACE-complete to decide whether a player has forced win from a certain board configuration.

Proof. Reduction from Bounded 2CL. Given a planar constraint graph made of AND, OR, SPLIT, CHOICE, and VARIABLE vertices, we can construct a corresponding Reversi game, as described above. The reduction may be done in polynomial time: if there are k variables and l clauses, then there need be no more than (kl)2 crossover gadgets to connect each variable to each clause it occurs in, all other aspects of the reduction are

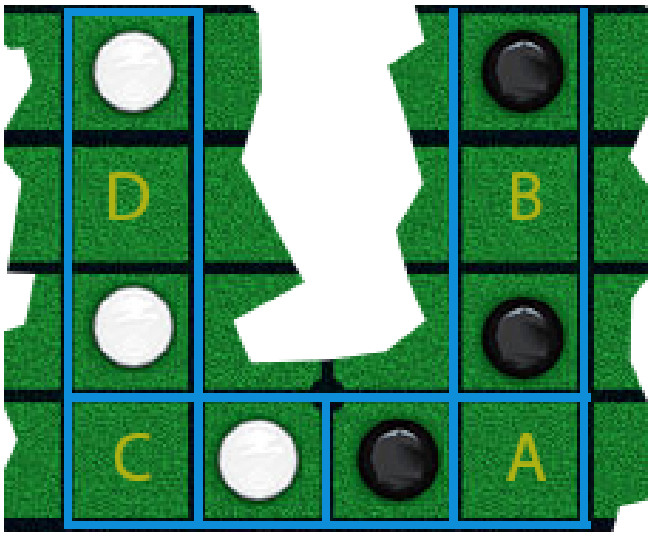


Figure 2.5: VARIABLE gadget

equally obviously polynomial. Reversi is definitely PSPACE-hard as it will end after a polynomial number of moves. So Reversi is PSPACE-hard and hence PSPACE-complete. □

2.2 Peg Solitaire

2.2.1 Introduction

The hardness of generalized versions of games have been studied in past. The problem of determining whether a particular player can win is found to be EXP-TIME complete for checkers, chess, etc. It is PSPACE-complete for games like amazons, konane. Peg Solitaire is already proved to be NP-complete by Ryuhei Uehara, Shigeki Iwata[4] by reducing from a variation of the Hamiltonian cycle problem. In this section we will be proving Peg Solitaire is NP-complete by using Bounded NCL which is much simpler to understand.

Peg solitaire is a board game which generally begins with pegs in every location on the board except for one location which is left empty (or a hole). Figure 2.8 represents the initial board. If in some row or column there are two adjacent pegs next to a hole, then the peg can jump over peg adjacent to it and occupies the hole. In this move the adjacent peg is removed from the board. Fig 2.9 represents a single move. The goal of the game is to remove every peg but one.

2.2.2 Basic Gadgets

The number of moves on a n x n board is polynomially bounded by n. Hence Peg Solitaire is a bounded single player game. we will reduce Erik's bounded NCL to peg solitaire. The bounded NCL consists of 4 gadgets which are as follows:

1. OR
2. AND
3. SPLIT
4. CHOICE

and few utility gadgets (wire, turns, shifter). In the following sections we will denote player A as white player and player B as black player. White player can win if it can remove all the pegs on the board but one. The cells marked with alphabets are empty and any of the player can tap on it and make a move if it is valid.

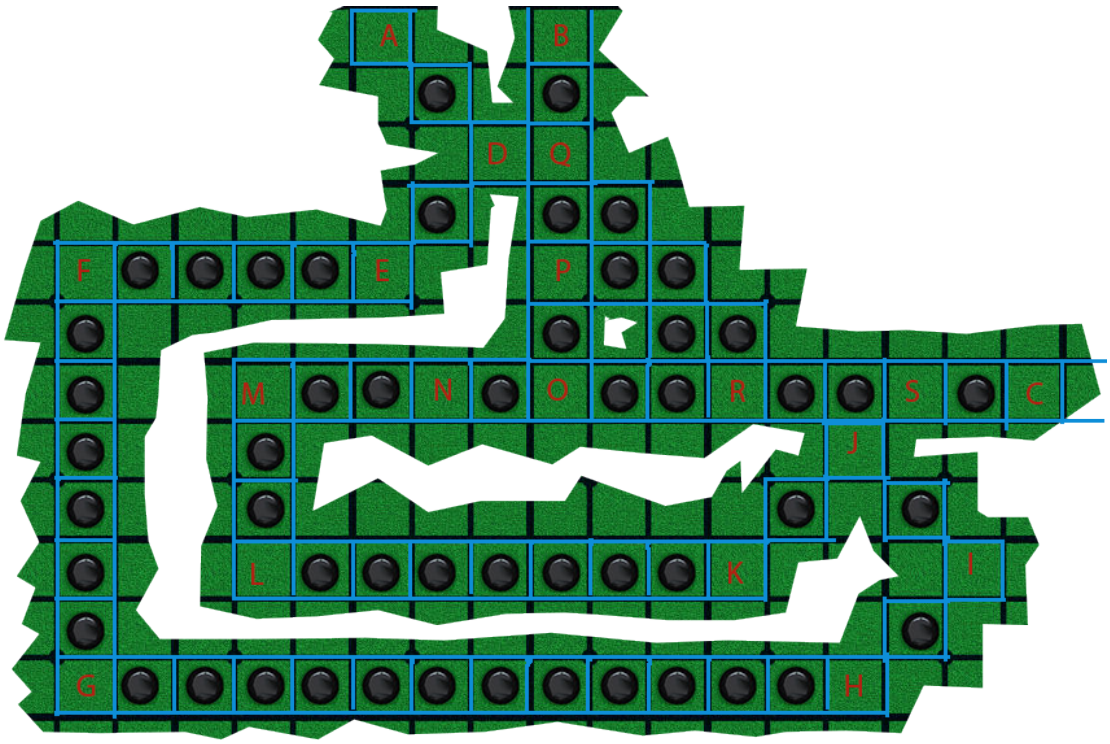


Figure 2.6: CHOICE gadget

2.2.2.1 UTILITY

Figure 2.10 shows the utility Gadget. These gadgets are used to connect to vertices(gadgets). A CHAIN is used to make a linear movement from one gadget to the other. A COLUMN SHIFT is used to shift the movement by one column to the right. A mirror image of the gadget can shift the movement by one column to the left. A TURN is used to take turns whenever required. The TURN gadget in the figure turns the movement to left, a mirror image of the same figure turns the movement to right.

2.2.2.2 OR

Lemma 2.2.1. In an OR gadget a player can put a marble on cell marked A or B then it can move to cell marked C via D.

Proof. Figure 2.11 shows the OR Gadget. If a player can put a marble on cell marked A, then it can jump over the peg and reach cell D. After this move, it can jump over the peg to reach C. Or if the player can put a marble on cell marked B, then it can jump over the peg and reach cell D. After this move, it can jump over the peg to reach C. □

2.2.2.3 AND

Lemma 2.2.2. In an AND gadget the player can reach C if and only if the player can reach both the cells marked A and B.

Proof. Figure 2.12 shows the AND Gadget. If the player can reach A, then it can reach F via D. Then there is no movement possible from D as there is no adjacent peg to F. An adjacent peg can be brought to cell E if the player can reach B. Then the peg on cell E can jump over peg on cell F to reach cell G and then to C. Or if the player reaches cell B then it can move to E by jumping over the adjacent peg. At this movement no adjacent peg is available, so no move is possible. If the player can reach A then it jumps over cell D and then to F. Then the peg on cell E can jump over peg in cell F to reach cell G and then to C. Note that the peg on cell F can jump over E, but after making that move any further move is not possible. □

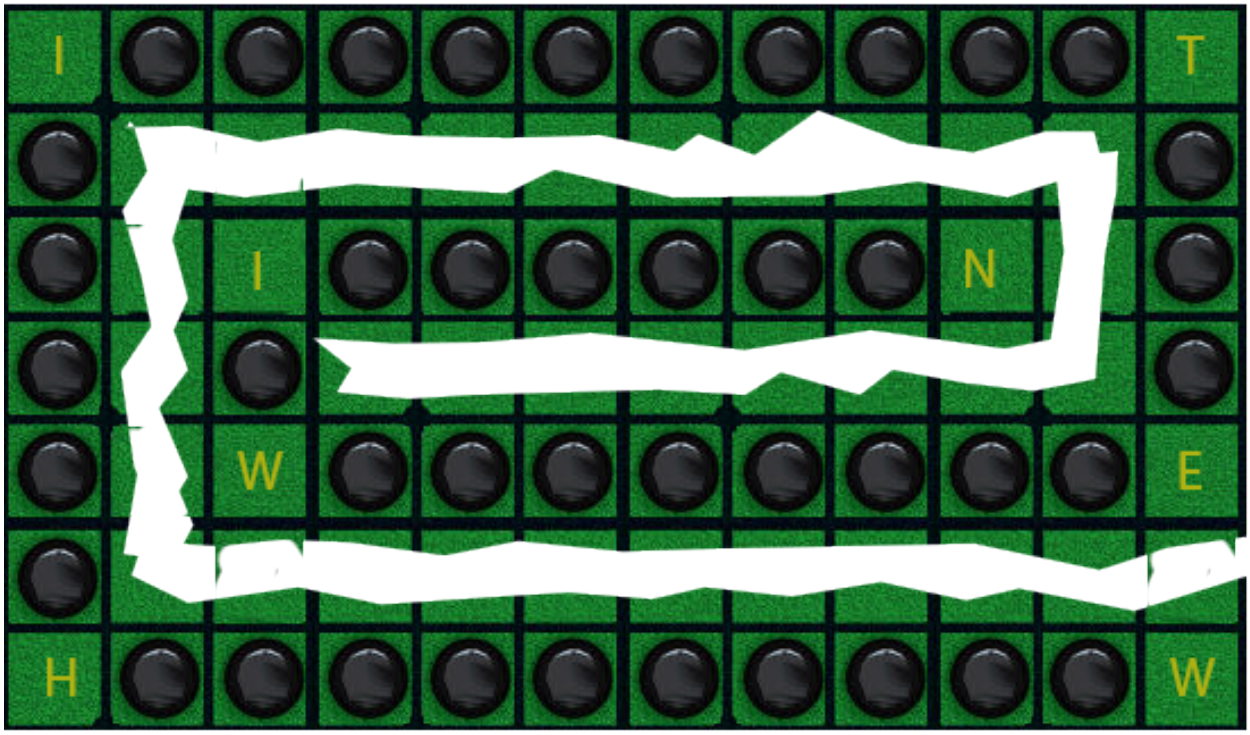


Figure 2.7: VICTORY gadget

2.2.2.4 SPLIT

Lemma 2.2.3. A player can reach B and D if it can reach cell A.

Proof. Figure 2.13 shows the AND Gadget. whenever the player is able to reach cell A, a free peg will be instantly available on cell C. If the player can reach cell A then it can reach cell B by making a series of moves. Then the player may use the free stone available to reach D by making a series of moves. So if the player can reach A then it reaches both B and D. □

2.2.2.5 CHOICE

Lemma 2.2.4. A player can reach either cell B or C but not both, if it can reach cell A.

Proof. Figure 2.14 shows the CHOICE Gadget. If the player can reach cell A it can move to cell E. After that move it can reach either B or C. It can't reach both B and C because after making a move the pegs are removed from the board. Backward movement is not possible and hence once the player has reached cell B, it is not possible to backtrack to cell E and then move to cell C and vice versa. so once a player has reached cell A it can leave the gadget via cell B or C but not both. □

2.2.2.6 VICTORY

The AND gadget corresponding to the target edge in the NCL graph will be connected to the VICTORY gadget shown in Fig. 2.15. If the player can activate this AND gadget, then it can remove all the pegs but one from the board and win the game. If the player is able to perform this action the corresponding target edge can be reversed in the NCL graph.

Theorem 2. Given a n x n Peg Solitaire Board it is NP-complete to decide whether a player has forced win from a certain board configuration.

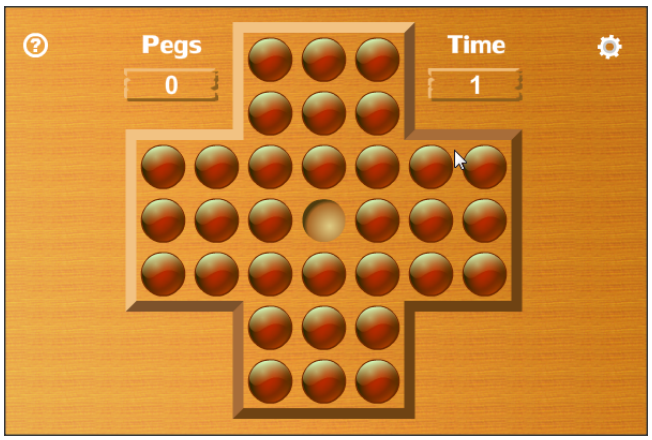


Figure 2.8: Initial board position

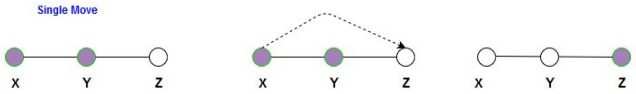


Figure 2.9: single move

Proof. Reduction from Bounded NCL. Given a planar constraint graph made of AND, OR, SPLIT, CHOICE vertices, we can construct a corresponding peg solitaire board game, using the gadgets above. The utility gadgets are used to connect the basic gadgets required to build the board. Initially, for every variable in the Bounded NCL graph we provide a peg at every corresponding cell in the board. The player can start on some gadget input corresponding to the location of the single loose edge and reach the VICTORY gadget just when the target edge in the constraint graph may be reversed. Therefore, Peg Solitaire is Np-hard.

Peg Solitaire is clearly in NP: there are only a linear number of pegs that can be removed and therefore a potential solution may be verified in polynomial time. □

2.3 Peg solitaire on special Graph Classes

Definition 2.3.1. State Its an ordered pair S = (V, E) which consists of set of vertices V and a set of edges E. Each vertex is numbered, which either contains a peg or is empty. A vertex v can be represented as v_r where
v vertex number
r = 1 vertex contains a peg
r = 0 vertex is empty
There exists an edge between every adjacent vertex. Edge E is undirected can be represented as (x_p, y_q).
Example:
S = (V, E)
V = {1_1, 2_1, 3_0,..,12_0,.., 49_1}
E = {(1_1, 2_1), (1_1, 9_1), (1_1, 8_1), ,(12_0,13_1), ,(49_1,48_1)}
Figure 2.16 represents a state.

Definition 2.3.2. Starting State A starting state S(V, E) contains pegs in every vertex except one and set of edges E.
S = (V, E)
V = {1_1, 2_1, 3_1,..,25_0.., 49_1} E={ (1_1, 2_1), (1_1, 9_1), (1_1, 8_1), ,(49_1,48_1)}
Figure 2.17 represents a Starting state.

Definition 2.3.3. Terminal State A state T(V, E) where all the vertices are empty except one and set of edges E. T = (V, E) V = 1_0, 2_0, 3_0,..,41_1.., 49_0

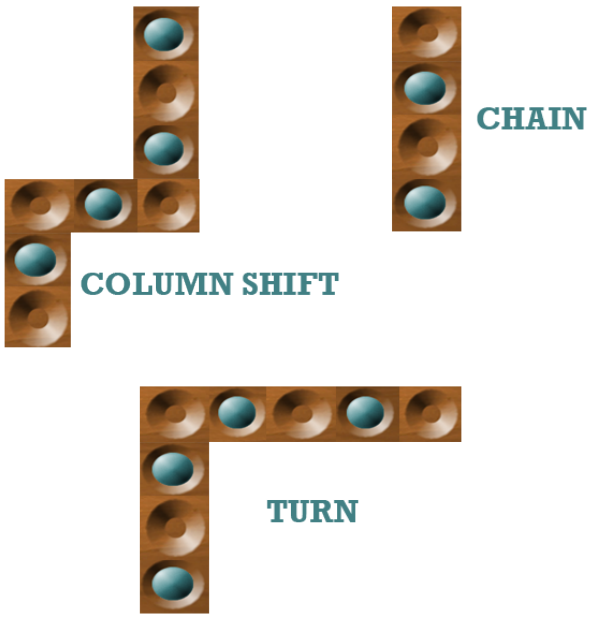


Figure 2.10: utility gadget

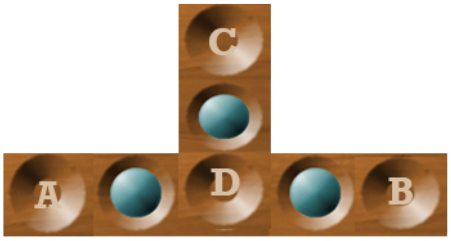


Figure 2.11: OR gadget

E= $(1_0, 2_0)$, $(1_0, 9_0)$, $(1_0, 8_0)$, , $(41_1, 42_0)$,$(49_0,48_0)$
Figure 2.18 represents a Terminal state.

Definition 2.3.4. Fully Filled State A state F (G, H) having pegs at every vertex and a set of edges H.
Example:
F = (G, H)
G = $\{1_1, 2_1, 3_1,..,12_1,.., 49_1\}$
H = { $(1_1, 2_1)$, $(1_1, 9_1)$, $(1_1, 8_1)$, $(12_1,13_1)$,$(49_1,48_1)$}
Figure 2.19 represents a Fully Filled state.

Definition 2.3.5. Move A move on a state S(P, Q) can be represented as M: $a_1 \rightarrow b_1 \rightarrow c_1$, if edge $e_1(a_1, b_1)$ $\in$ Q and $e_1(b_1,c_1) \in$ Q
S_i(V,E) V=$\{a_1,b_1,c_0,.\}$ E =$\{(a_1,b_1), (b_1,c_0),..\}$
Move m: a1 $\rightarrow$ b1 $\rightarrow$ c0
S_{i+1}(V,E) V= $\{a_0, b_0, c_1,.\}$ E = $\{(a_0,b_0), (b_0,c_1),..\}$

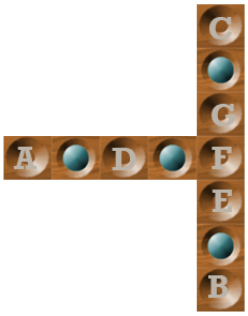


Figure 2.12: AND gadget

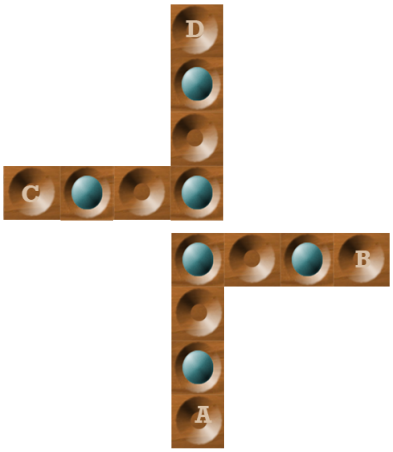


Figure 2.13: SPLIT gadget

Definition 2.3.6. Association A terminal state T (X, Y) is associated with starting state S (V, E) if T can be obtained from S by series of moves.

Definition 2.3.7. Freely Solvable A graph G (V, E) is freely solvable if for all starting states S of graph G (V, E), there exists an associated terminal state T (X, Y) consisting of a single peg.

Definition 2.3.8. K Solvable A Fully filled state F (G, H) is k-solvable if there exists a starting state S (P, Q) of F (G, H) such that there exists an associated terminal state T (X, Y) consisting of k non-adjacent pegs.

Definition 2.3.9. Distance 2 Solvable A Fully filled state F (G, H) is Distance 2-solvable if there exists a starting state S (P, Q) of filled state F (G, H) such that there exists an associated terminal state T (X, Y) consisting of two pegs that are distance 2 apart.

2.3.1 Graphs Studied

- Star $K_{1,n}$
- Path P_{2n} ,P_{2n+1}
- Cycles C_{2n},C_{2n+1}

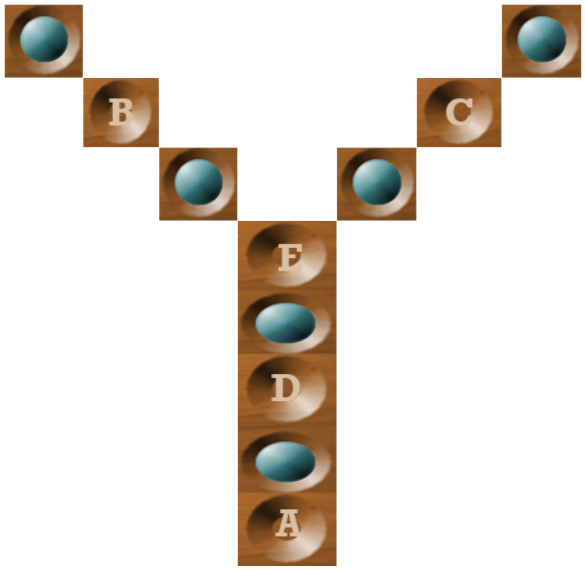


Figure 2.14: CHOICE gadget

- Complete Graph K_n
- N-ary Tree
- Petersen Graph

Theorem 3. The star $K_{1,n}$ is (n − 1)-solvable.[5]
If n > 1, then P_{2n} is not freely solvable.[5]
P_{2n+1} is not solvable.[5]
P_n is solvable iff n=3 or n is even.[5]
P_n is solvable iff n=3 or n is even.[5]
P_n is 2 distance solvable in all other cases.[5]
C_n is freely solvable if n is even or n=3.[5]
C_n is 2 distance solvable in all other cases (n is odd and n> 3).[5]
K_n is freely solvable[5]

2.3.1.1 The star $K_{1,n}$ is (n −1)-solvable.

Basically there are only 2 choices for the initial hole. One is the pendant vertices other being the center vertex. If the initial hole is at pendant vertex, then we can remove only one peg from the graph. If the hole is at center, then there is no move. Figure 2.22 represents the choices.

2.3.1.2 If n > 1, then P_{2n} is not freely solvable.

- In a path, any move will result in an edge having holes on both end points. Figure 2.23 depicts such a move.
- To solve this path, we should be able to fill the holes of the empty bridge.
- If the initial hole is at vertex 0 then it is not possible to fill the bridge and hence P_{2n} is not freely solvable.

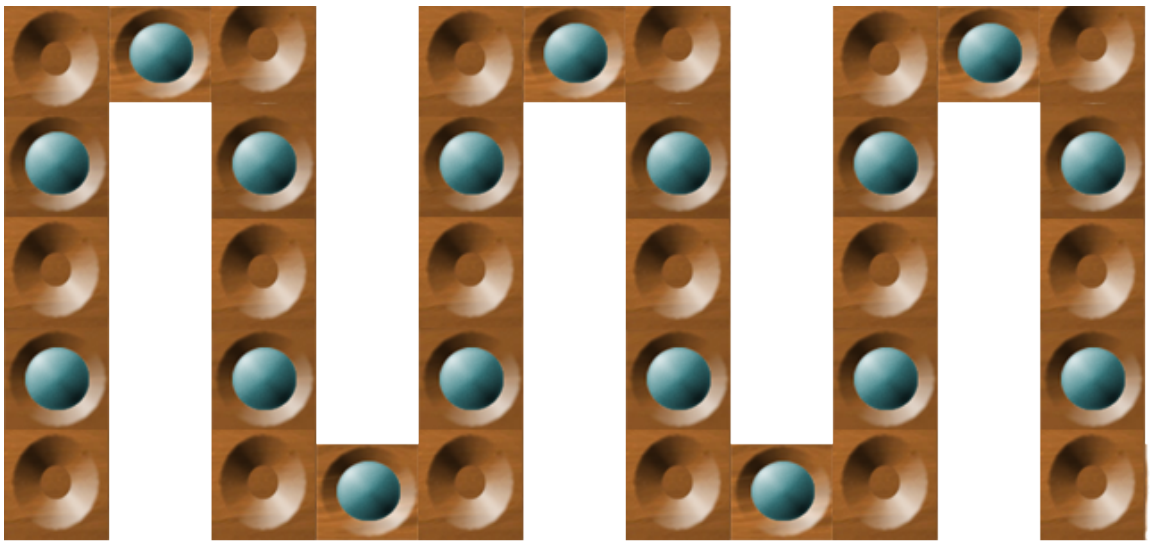


Figure 2.15: VICTORY gadget

2.3.1.3 P_{2n+1} is not solvable.

- The initial hole divides the pegs into two groups. Both the groups have either even number of pegs or odd. In a single move two pegs are removed from one group and one peg is added to another group.
- After the first move one of the group is even(2k) and the other is odd(2k+1). Every time two pegs are removed from the odd side an empty bridge is formed and the size of vertices on odd side goes down by 2. After K such moves there will be an empty bridge having only one peg on its one side, which is not solvable. Figure 2.24 represents the moves discussed above.

2.3.1.4 P_n is solvable iff n=3 or n is even.

- P_n where n is even
- Place the hole at n-1
- Move n-3 over n-2 to n-1
- Move n over n-1 to n-2
- The problem size has reduced to P_{n-2}. Figure 2.25 follows the steps described above.

2.3.1.5 P_n is 2 distance solvable in all other cases.

- P_n is odd.
- Place the hole at n-1.
- Move n-3 over n-2 to n-1.
- Move n over n-1 to n-2.

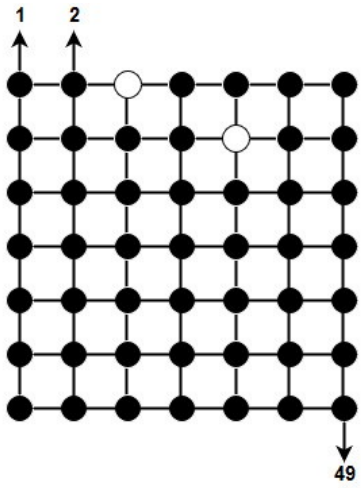


Figure 2.16: state

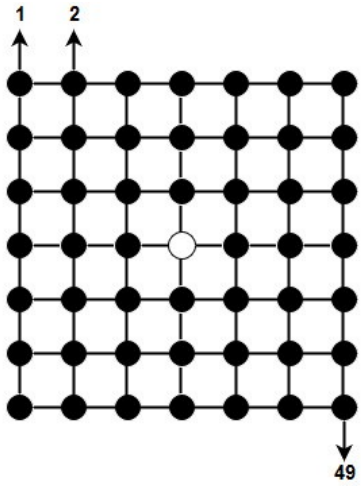


Figure 2.17: Starting State

- The problem size has reduced to P_{n-2}.
- As the number of vertices are odd, the last empty bridge cant be filled to solve the path. Figure 2.26 follows the steps described above.

2.3.1.6 C_n is freely solvable if n is even or n=3.

- Place the hole at n
- Move 2th peg to nth vertex
- Move 4th peg to 2th vertex
- The problem has reduced to Pn-2 which is solvable when the hole is placed at (n-1)th vertex. Figure 2.27 follows the steps described above.

2.3.1.7 C_n is 2 distance solvable in all other cases (n is odd and n>3).

- Place the hole at n
- Move 2th peg to nth vertex

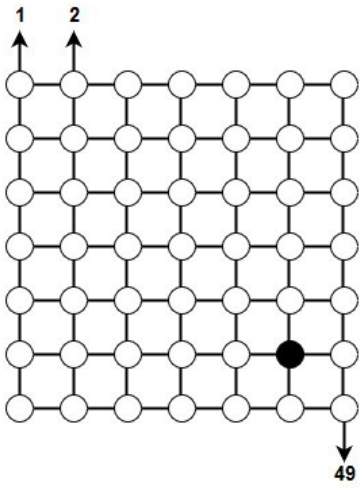


Figure 2.18: Terminal State

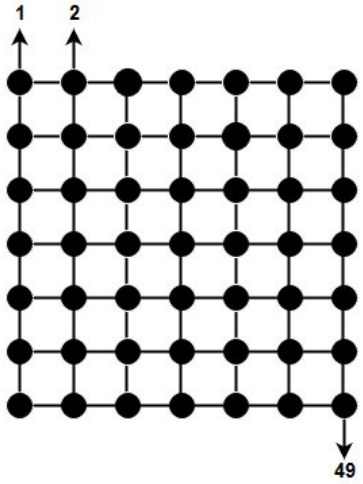


Figure 2.19: Fully Filled State

- Move 4th peg to 2th vertex
- The problem has reduced to P_{n-2} which is solvable when the hole is placed at (n-1)th vertex. Here n-2 is odd. As we know P_n where (n-2) is odd is 2 distance solvable, then the cycle C_n where n is odd is 2 distance solvable Figure 2.28 follows the steps described above.

2.3.1.8 K_n is freely solvable[5]

- Place the hole at n.
- Move 2th peg to nth vertex.
- The problem has reduced to K_{N-1}. Figure 2.28 follows the steps described above.

2.3.1.9 N-ary tree of size N is not freely solvable.

- A path is a tree. P_{2n} is not freely solvable. So N-ary tree is not freely solvable.

2.3.1.10 Lower bound on N-ary tree of size N.

- Here we give a lower bound on number of pegs that can be removed.

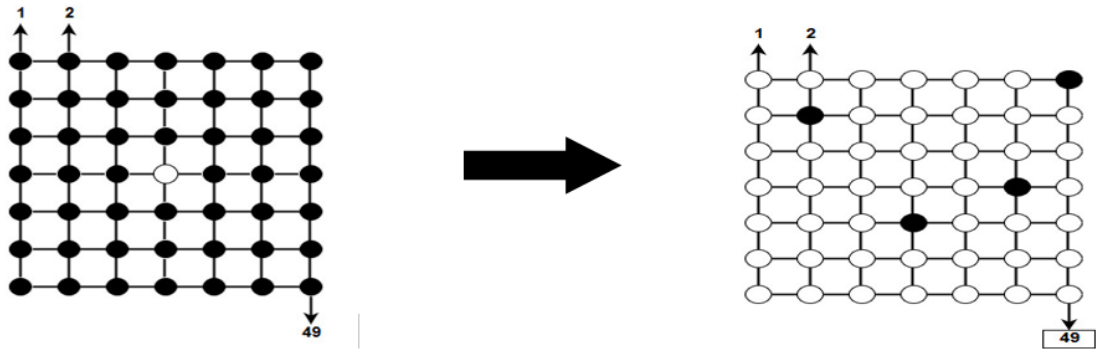


Figure 2.20: K Solvable

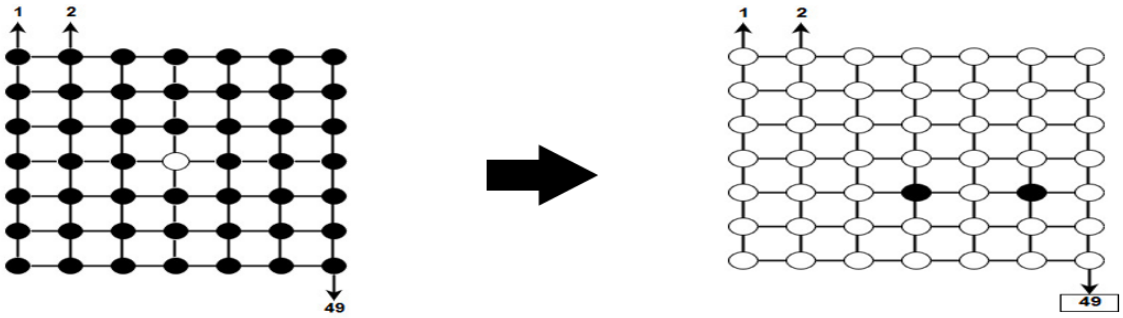


Figure 2.21: Distance 2 Solvable

- Let D be the diameter of the tree T.
- The diameter D can be considered as a path of length P_{2k} if $\mid D \mid$ is even or P_{2k+1} if $\mid D \mid$ is odd.
- We know P_{2k} is solvable. So we can remove at least (2k-1) peg form the tree.
- We know P_{2k+1} is 2 distance solvable. So we can remove at least remove (2k+1)-2 pegs from the tree. Figure 2.29 follows the steps described above.

2.3.1.11 Observation based on Hamiltonian Path.

- A graph G(V, E) is solvable if it contains a Hamiltonian path of size (2k). We know P_{2k} is solvable and a Hamiltonian path covers every vertex exactly once.
- A graph G(V, E) is Distance 2 solvable if it contains a Hamiltonian path of size (2k+1). We know P_{2k+1} is Distance 2 solvable and a Hamiltonian path covers every vertex exactly once.

2.3.1.12 Petersen Graph is solvable

- In the mathematical field of graph theory, the Petersen graph is an undirected graph with 10 vertices and 15 edges.
- Peterson Graph is solvable.
- It contains a Hamiltonian path of even size. Figure 2.30 represents the path.

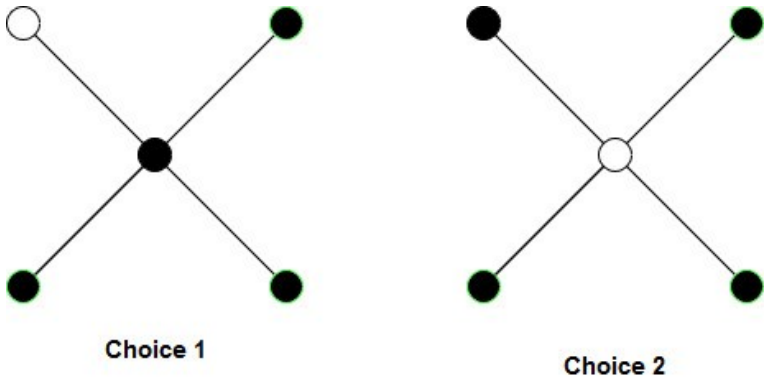


Figure 2.22: choices

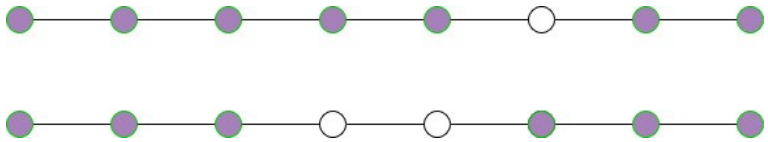


Figure 2.23: Even Path Length

2.3.1.13 Lowering the upper bound of number of holes required to solve a Tree T(V, E).

Problem Definition

Given a Tree T (V, E) find the minimum number of holes required to solve this tree.

Figure 2.31 represents the algorithm and Fig. 2.32 represents the flowchart.

Example

The Tree T (V, E) given in Fig.2.33 contains 37 pegs.

Find the minimum number of holes required to solve the tree.

Figure 2.34 represents the tree along with the diameter found after every recursive call. Table 2.1 represents the diameters removed on following the algorithm given in Fig. 2.31.

Diameter	Size	Pegs Removed	Pegs Left
1	15	13	2
2	5	3	2
3	5	3	2
4	5	3	2
5	4	3	1

Table 2.1: Diameters removed

- Holes Required = holes + pegs
- holes = Number of diameters
- pegs = pegs which cant be removed + pegs which are left from the diameter
- In this example,
- holes = 5 + (9 + 3) -1 = 16
- The upper bound of number of holes required is (37-1) = 36 and we have reduced it to 16. we have reduced more than half of upper bound.

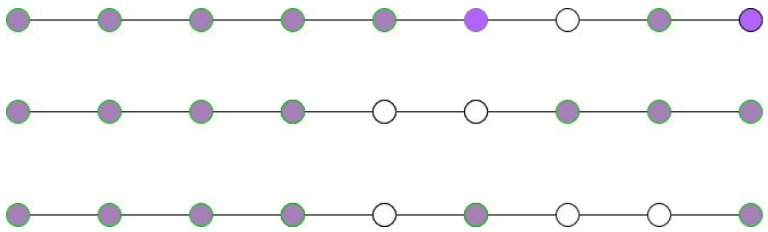


Figure 2.24: Odd Path Length

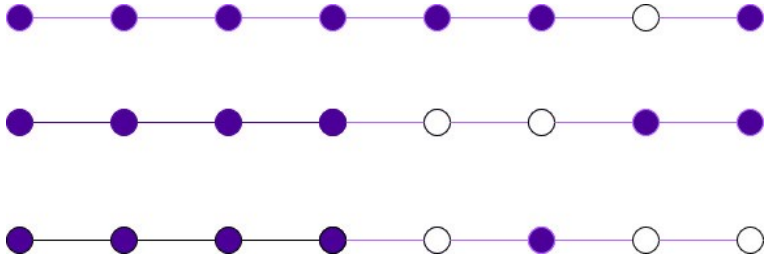


Figure 2.25: Solvable Path

2.4 Badland

2.4.1 Introduction

Badland is a single player mobile game which consists of several levels. It features a black minion which the player has to guide to reach the end of the level. Each level is full of obstacles like lasers which can kill the minion if it comes in contact with it, big stones which can fall on the minion and crush it down, wheel grinders which can grind the minion if it comes in contact with it and so on. The level also consists of Power-Up like speeding or slowing down the movement of the minion, expand or shrink the minion, clones the minion to multiple minions so that the player is alive until and unless all the clones die and many more. If the player dies while overcoming an obstacle, then the player respawns at the last checkpoint which it has crossed in that level. There is no time bound to complete a level. Figure 2.35(a) shows the portal from which the minion spawns at the start of the level. Figure 2.35(b) displays the lasers which can kill the minion. The lasers switch on and off periodically to allow a gap for the minion to move over it. Figure 2.35(c) features the death balls which can end the life of the minion if it comes in contact with it. Figure 2.35(d) shows the two grinders which rotates in high speed and prove to be an obstacle for the minion. The yellow colored object in the Fig. 2.35(e) is a power up which can expand the size of the minion. There are two different size of yellow colored objects in Fig. 2.35(f), the large one expands the size of the minion and the smaller one shrinks the size the minion. Apart from this Power-Up the 3 black colored objects can clone the minion into multiple minions as a result increasing the lifespan of the minion in that level. We will prove that Badland is NP-Complete which don't appear in any previous research work.

3-SAT

Given a Boolean expression consisting of clauses, where every clause is of size three and every literal in a clause is ORed and every clause is ANDed. Is there any such assignment(TRUE/FALSE) of literals such that the whole expression evaluates to TRUE.

2.4.2 Version of the Game

The version of game we are going to consider is played in a single screen and it has one level. The level consists of objects like High speed fans, death ball, teleports and switches. The player can only move in a direction towards the end of the level.

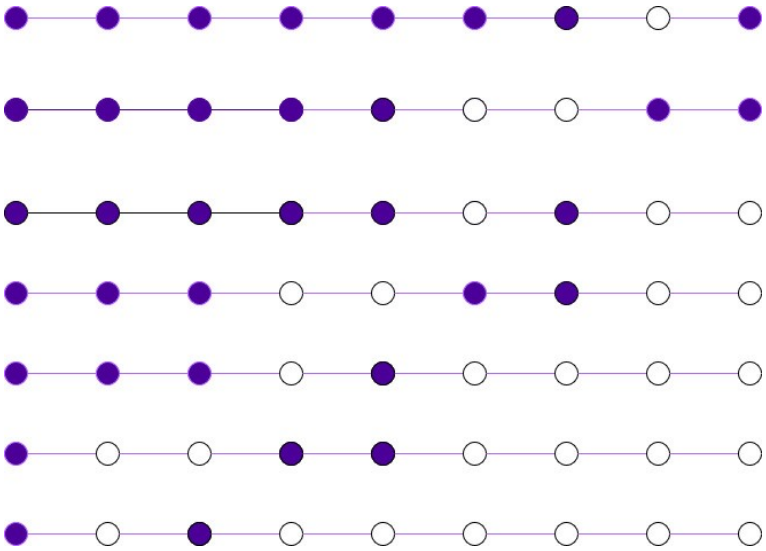


Figure 2.26: 2 Distance Solvable Path

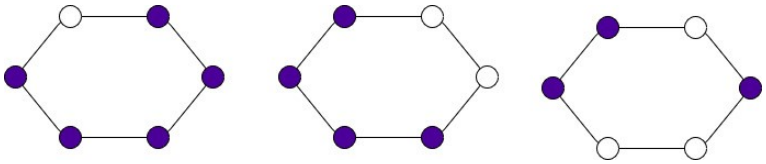


Figure 2.27: Cycle

2.4.3 Decision Problem

Given a complete level in a single screen can the player reach the end of this level facing the obstacles on the path.

2.4.4 Theoretical Problem definition corresponding to a single level

Given a directed Graph with out-degree of every node ∈ {1,2} and in-degree ∈ {1, 2, 3,..}. Disjoint subsets S = {s_1, s_2, s_3,., s_k} of edges. Given a start vertex s and a target vertex t. Reach t from s and traverse at least one edge from every subset of S.

Theorem 4. Given a complete level of Badland on a single screen it is NP-complete to decide whether the player can reach the end of this level facing the obstacles on the path.

Proof. Reduction from 3-SAT problem. Given a 3-sat instance we will construct a badland level corresponding to this instance. If a player can win the game, the assignments of true/false for a variable can then be read from the switches, which will satisfy the given instance. so it is NP-hard.

Badland is clearly in NP: The player can guess a path and verify it by playing the level in polynomial time. Badland is NP-hard and is in NP class, Hence Badland is NP-complete.

□

3-SAT Instance

F (x, y, z) = (x V y V z) ˆ (x 'V y 'V z) ˆ (x V y V z ')

We will reduce this instance to a level of badland.

Badland Level

A single level basically consists of two types of gadgets:

1. Variable Gadget

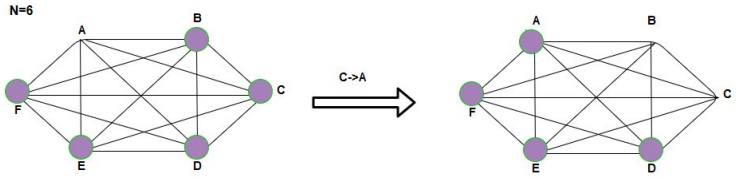


Figure 2.28: Cycle

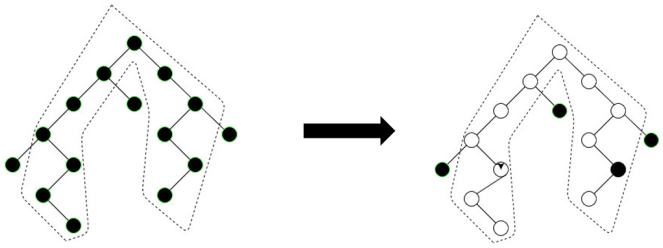


Figure 2.29: Lower Bound on Tree

2. Clause Gadget

1. **Variable Gadget**
   Each variable gadget consists of 3 portals. The player spawns from the portal at the top and move out of this gadget by one of the portals at the bottom. Figure 2.36 represents the variable gadget.

   Correspondence with 3-SAT
   If this variable gadget corresponds to variable X and the player leaves this gadget by the left portal that means X is set to TRUE and vice versa

2. **Clause Gadget**
   Each clause gadget consists of 6 portals, 3 switches, 1 high speed fan, few death balls.

   The player spawns form one of the portal at the top and tap the corresponding switch which switches off the high speed fan and move out of this gadget by the portal at the bottom. Figure 2.37 represents the clause gadget.

   Correspondence with 3-SAT
   If this gadget is visited once then this clause which is a part of the function evaluates to true.

Single pass of the Level

Figure 2.38 represents a single level of Badland.

- The game starts from the portal marked as 1. The objective is to reach the green colored portal at the bottom right corner.
- The players movement inside a gadget is already explained in the gadgets section. If the player enters a portal marked p the player will be teleported to portal marked (p+1).
- After the player has played through all the variable gadgets and clause gadgets the player has to pass through a tunnel which has death balls and high speed fans. If any one of the high speed fans is switched ON, then the fan will force the player to death balls and the game will get over. If all the fans are switched OFF then the player can pass through the tunnel and reach the green portal and win the level.

Construction of the Level

This level is created using the variable and clause gadgets discussed above. For every variable in the function

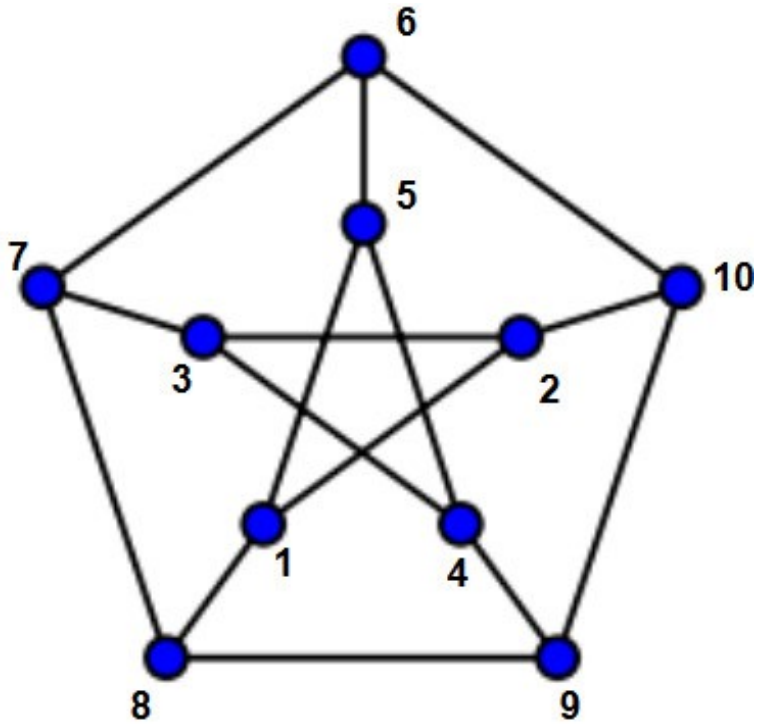


Figure 2.30: Peterson Graph

F (x, y, z) there is a variable gadget and for every clause in the 3-Sat instance there is a clause gadget. The portals are connected in such a way that a variable gadget can be visited only once. When a variable gadget is visited then it is so connected that it will visit all the clauses having this variable. A tunnel is created with high speed fans and death balls which corresponds to clauses in the F (x, y, z). For every clause a high speed fan is there. The level ends at the end of this tunnel.

The following algorithm lowers the upper bound of the number of holes.
Algorithm

```
solve(Tree T){
        Diameter D=findDiameter(T);             //findDiameter(tree T) finds a diameter of the given
Tree T
        if(D==null)                     //if the tree do not contain a diameter then end the function
                end
        else{
                if(D.size is even)
                        pegs-=(D.size-1)        //p2k is solvable
                else
                        pegs-=(D.size-2)//p2k+1 is Distance 2 solvable
                holes++;                //every diameter requires a hole
                ∀v ∈ D
                        Tree subT=findSubTree(child(v∉D));      //findsubTree(Node n) returns a
subtree rooted at n
                        if(subT.size>=2){       //subtree size of 1 can't be a candidate
                                solve(subT);    //recur for the subtree found
        }
}
```

Figure 2.31: Algorithm

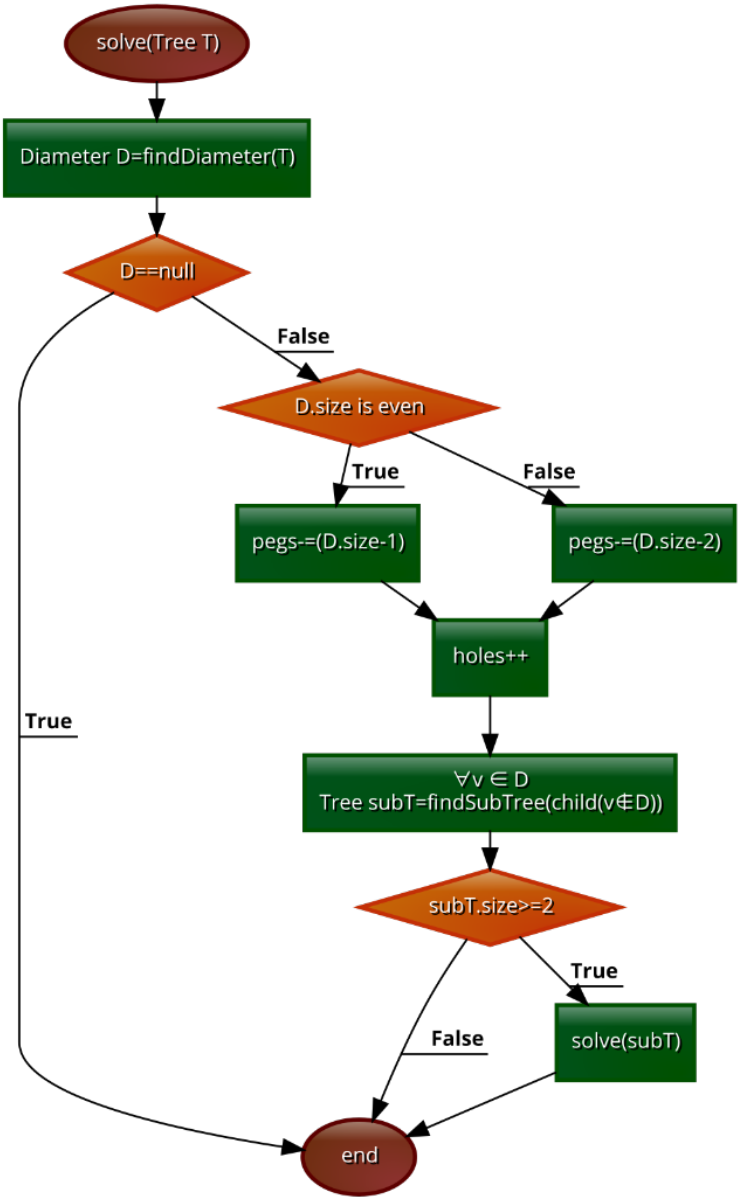


Figure 2.32: Flowchart

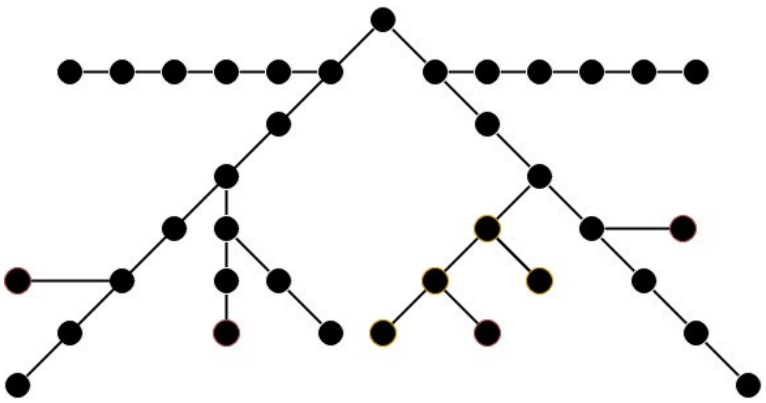


Figure 2.33: Tree

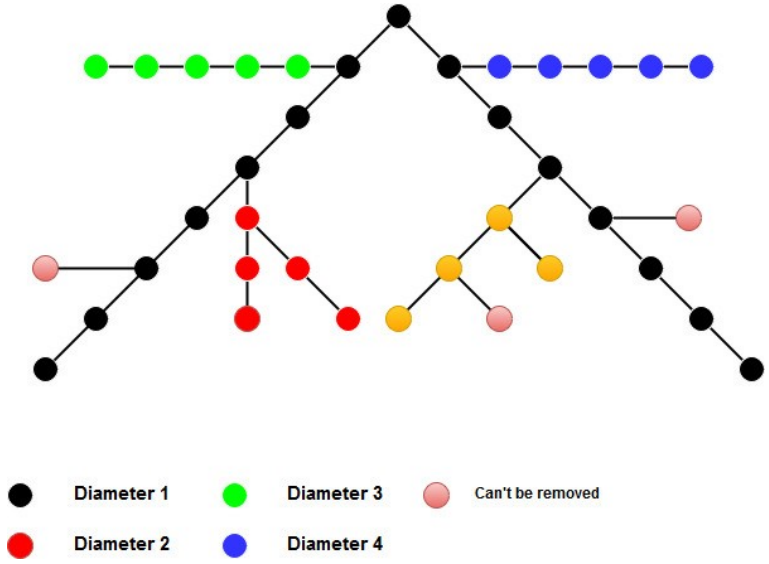
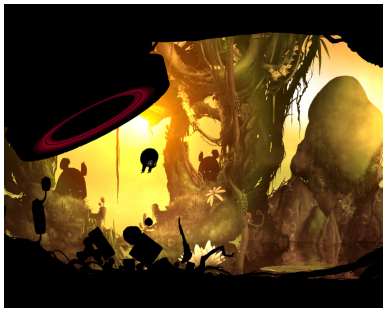
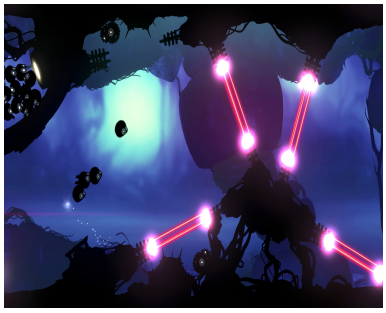


Figure 2.34: Tree with diameter



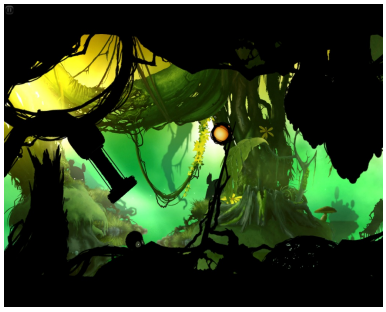
(a) Spawn Point



(b) Lasers

(c) Death Balls

(d) Grinders



(e) Size Power-Up

(f) Clone Power-Up

Figure 2.35: Screenshots of Badland

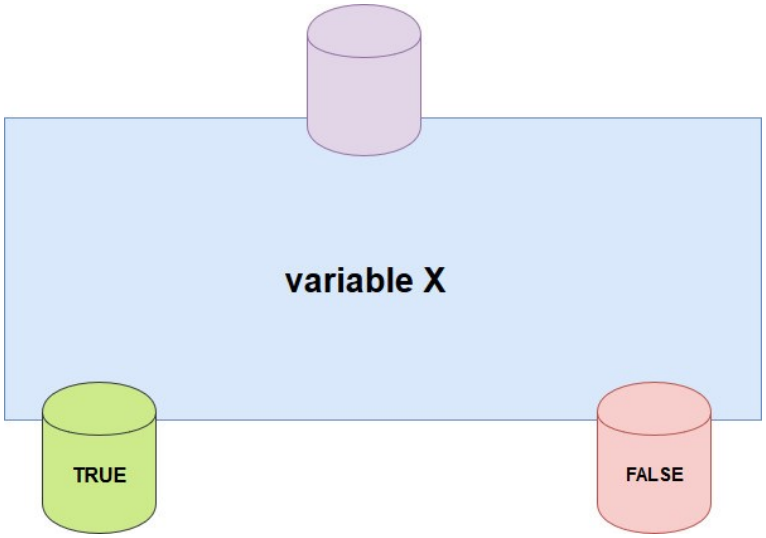


Figure 2.36: Variable Gadget

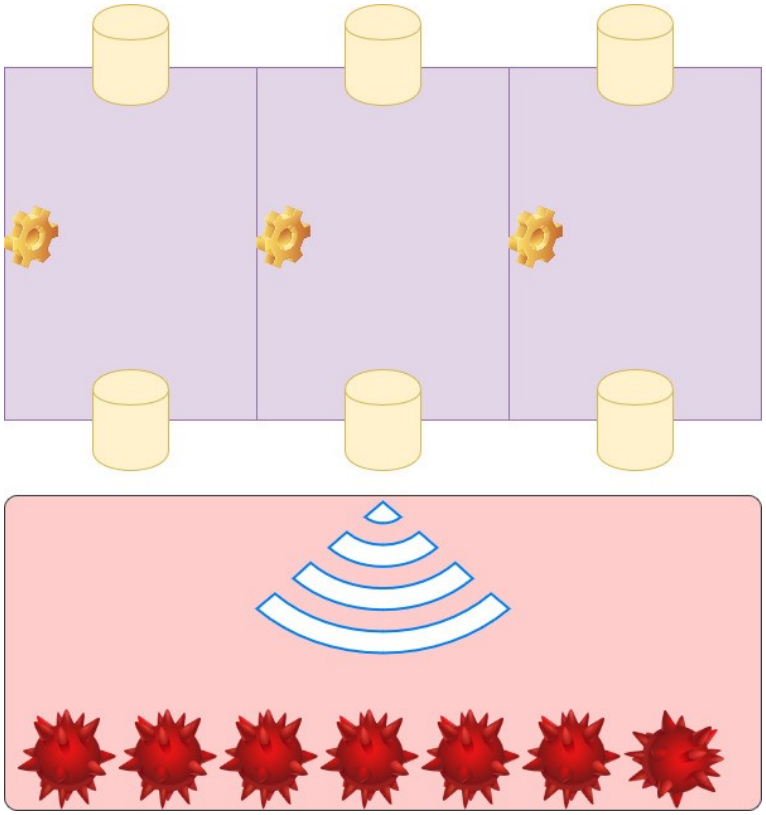


Figure 2.37: Clause Gadget

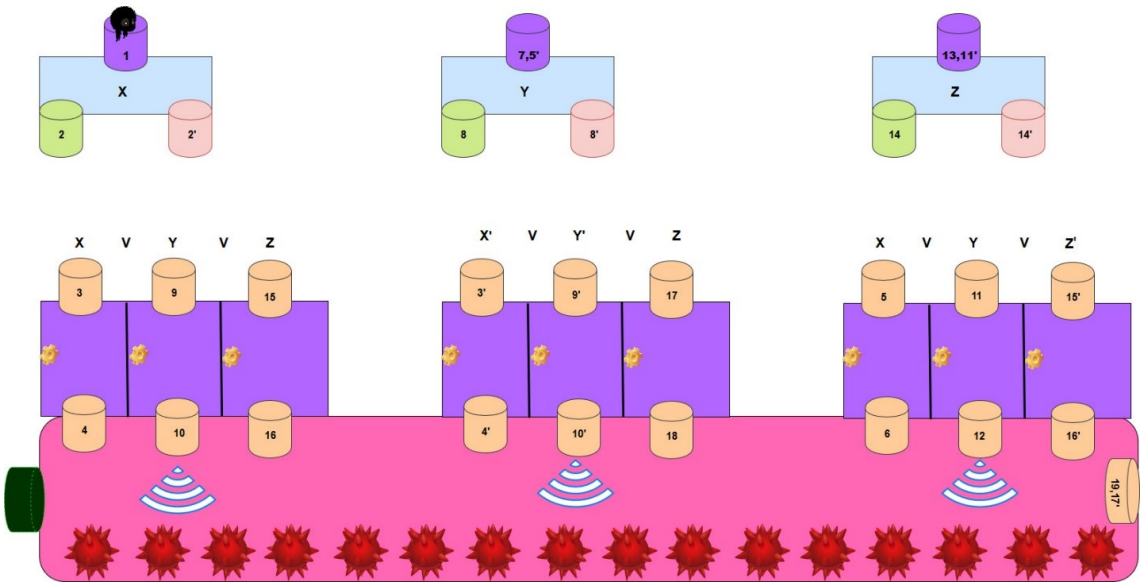


Figure 2.38: Level

Chapter 3

Graph Sampling

3.1 Introduction

A graph can represent many real-world complex systems like biological networks, communication networks and information networks. The networks are quite large and graph mining algorithms such as graph Partitioning, graph clustering, etc. are computationally expensive and generally do not scale well to very large graphs. To overcome this very challenge graph sampling provides a feasible and affordable solution for the analysis of large graph. A sampled subgraph is capable of experimentation and realistic simulations even before deploying new systems and protocols. In order to make accurate analysis on such large graphs, there is a need of sampling methods which can sample a representative subgraph that represents the same properties (e.g., degree distribution, clustering coefficient, average shortest path length, largest connected component size) as the original graph. It is not feasible to analyze the characteristics of a large graph. It can be estimated via a sample which closely represent the original graph.

Analyzing a subgraph of a large graph is convenient and relatively easy. But creating a subgraph is a challenge. Several sampling algorithms were proposed in literature which includes Random Node Sampling[6], Random Edge Sampling[6], Random Walk Sampling[6], Tightly Induced Edge sampling(TIES)[7]. All of them except TIES have certain drawbacks and provides inconsistent results. TIES seems to work well on graphs but it's performance decreases on trees. Our work shows that Tightly Induced BFS and KARGER sampling algorithm works well on trees.

3.2 Problem definition

Let G (V,E) represent the graph, where V is set of nodes and E is set of edges in the graph. Edge e $\in$ E can be described as a pair of vertices (v_i, v_j) where v_i, $v_j \in$ V. Given a sampling fraction δ, the objective is to produce a sample graph G_s (V_s,E_s) such that $\mid V_s \mid / \mid V \mid = \delta$ that retains the properties of original graph G. Note that here we assume the sampling algorithms have access to all the nodes and edges of the graph G. The properties like degree distribution, clustering coefficient, average shortest path length, largest connected component size are considered in this work.

3.3 Sampling Methods

The sampling methods that is used to sample a representative subgraph are as follows:

1. Random Node Sampling
2. Random Edge Sampling

3. Tightly Induced Edge Sampling
4. Forest Fire Sampling
5. Karger Sampling
6. Breadth First Search Sampling
7. Tightly Induced Breadth First Search Sampling

3.3.1 Random Node Sampling (RN)

The Random Node Sampling produces a sample by randomly selecting a subset of vertices and taking the induced subgraph.

Algorithm 1: Random Node Sampling

```
   Result: G_s(V_s, E_s)
1  sampleNodeCount = G.numberOfNodes * δ
2  listNodes = G.nodes()
   /* sampling nodes                                                    */
3  for i ← 1 to smpleNodeCount by 1 do
4  |  index=random.randint(1,size(listNodes))
5  |  sampleNodes.append(listNodes[index])
6  |  del listNodes[index]
7  end
   /* induction step                                                    */
8  G_s=Graph()/* creates an empty undirected unweighted graph           */
9  foreach node in sampleNodes do
10 |  G_s.addNode(node)
11 |  foreach neighbour in G[node].neighbours do
12 |  |  if neighbour in sampleNodes then
13 |  |  |  G_s.addEdge(node,neighbour)
14 |  |  end
15 |  end
16 end
```

3.3.2 Random Edge Sampling (RE)

The Random Edge Sampling produces a sample by randomly selecting a subset of edges from the original graph.

Algorithm 2: Random Edge Sampling

```
   Result: G_s(V_s, E_s)
1  sampleNodeCount = G.numberOfNodes * δ
2  listEdges = G.edges()
   /* sampling Edges                                                    */
3  G_s=Graph()/* creates an empty undirected unweighted graph           */
4  while G_s.numberOfNodes < sampleNodeCount do
5  |  index=random.randint(1,size(listEdges))
6  |  G_s.addEdge(listEdges[index])
7  |  del listEdges[index]
8  end
```

3.3.3 Tightly Induced Edge Sampling (TIES)

The Tightly Induced Edge Sampling produces a sample by randomly selecting a subset of edges from the original graph and taking the induced subgraph.

Algorithm 3: Tightly Induced Edge Sampling

Result: $G_s(V_s, E_s)$

```
1  sampleNodeCount = G.numberOfNodes * δ
2  listEdges = G.edges()
   /* sampling Edges                                                    */
3  G_s=Graph()/* creates an empty undirected unweighted graph           */
4  while G_s.numberOfNodes < sampleNodeCount do
5      index=random.randint(1,size(listEdges))
6      G_s.addEdge(listEdges[index])
7      del listEdges[index]
8  end
   /* induction step                                                    */
9  sampleNodes=G_s.nodes()
10 foreach node in sampleNodes do
11     G_s.addNode(node)
12     foreach neighbour in G[node].neighbours do
13         if neighbour in sampleNodes then
14             G_s.addEdge(node,neighbour)
15         end
16     end
17 end
```

3.3.4 Forest Fire Sampling (FFS)

The Forest Fire sampling starts by choosing a node uniformly at random and adding it to the sample. It then burns a fraction of edges attached to it. The fraction is a random number drawn from a geometric distribution with mean $p_f/(1-p_f)$. p_f is the burning probability. In this work $p_f = 0.7$, which means on an average each selected node burns 2.33 nodes from its neighbors. This above process is recursively applied for each burnt neighbor until our budget (i.e. sample size) is reached.

Algorithm 4: Forest Fire Sampling

Result: $G_s(V_s, E_s)$

```
sampleNodeCount = G.numberOfNodes * δ
G_s=G.copy()/* creates a copy of original graph */
seedNode=random.choice(G_s.nodes()) Queue=() /* creates a empty queue */
Queue.put(seedNode)
visited=set() /* creates empty set */
visited.add(seedNode)
while G_s.numberOfNodes > sampleNodeCount do
    if Queue is empty then
        seedNode=random.choice(G_s.nodes())
        Queue.put(seedNode)
    end
    currNode=Queue.get()
    if currNode is already burnt then
        continue
    end
    candidate=list()
    foreach node in G_s[node].neighbours(currNode) do
        if node not in visited then
            candidate.append(node)
        end
    end
    nodesToBeBurnt=random.geometric(0.7,1)
    selected=candidate if nodesToBeBurnt < len(candidate) then
        selected=random.sample(candidate,nodesToBeBurnt)
    end
    foreach node in selected do
        Queue.put(node)
        visited.add(node)
    end
    G_s.remove_node(currNode)
end
```

3.3.5 Karger Sampling (KS)

Karger Algorithm is used to find min cut of a graph. It is modified to suit our purpose. This is a contraction based algorithm where an edge is contracted to produce a new vertex labeled with the vertices of the edge, all the edges from the two vertices are added to this newly formed vertex. Please note if the edge e is (a, b) and there is an edge (a, c) and (b, c) both the edges will be added to the newly formed vertex. So this contraction of edges continues until a vertex meets the budget or no edge is left to contract. In the former case we take the induced subgraph formed by the vertex, in the latter case we take a portion of nodes from every vertex to meet our budget. This algorithm has not appeared in any previous research work.

Algorithm 5: Karger Sampling

```
Result: G_s(V_s, E_s)
sampleNodeCount = G.numberOfNodes * δ
groups=dictionary()
dGraph=G.copy()/* creates a copy of original graph */
groupId=-1
while true do
    /* pick random edge */
    edgeList=dGraph.edges() if edgeList is empty then
        break
    end
    redge=random.choice(edgeList)
    /* contract edge */
    u=redge[0]
    v=redge[1]
    neighbours=dGraph.neighbours(v)
    dGraph.remove_node(v)
    groups[u].add(v)
    if size(groups[u])≥ sampleNodeCount then
        groupId=u
        break
    end
    foreach node in neighbours do
        if u==node then
            continue
        end
        dGraph.add_edge(u,node)
    end
end
sampleNodes=list()
if groupId==-1 then
    /* taking a vertex */
    sampleNodes=groups[groupId] sampleNodes.append(groupId)
else
    /* taking a subset of vertices from every vertex */
    foreach key, value in groups do
        sampleNodes.append(key)
        nodesTobeTaken=δ * size(value)
        temp=random.sample(value,nodesTobeTaken)
        sampleNodes.append(temp)
    end
end
```

3.3.6 Breadth First Search Sampling

This is the same traditional BFS algorithm where a seed node is chosen at random and then nodes are discovered one by one and we keep adding the nodes to sample until a budget is reached.

```
    /* deleting vertices from sampleNodes if it exceeds sampleNodeCount                */
(37) if size(sampleNodes) ≥ sampleNodeCount then
(38)     for i ← sampleNodeCount + 1 to sampleNodeCount by 1 do
(39)         if G_s is a tree then
(40)             delete a leaf
(41)         else
(42)             delete sampleNodes[sampleNodeCount+1]
(43)         end
(44)     end
(45) end
    /* induction step                                                                  */
(46) G_s=Graph()/* creates a empty graph                                              */
(47) foreach node in sampleNodes do
(48)     G_s.addNode(node)
(49)     foreach neighbour in G[node].neighbours do
(50)         if neighbour in sampleNodes then
(51)             G_s.addEdge(node,neighbour)
(52)         end
(53)     end
(54) end
```

Algorithm 6: Breadth First Search Sampling

```
    Result: G_s(V_s, E_s)
(1)  sampleNodeCount = G.numberOfNodes * δ
(2)  G_s=Graph()/* creates an empty undirected unweighted graph                      */
(3)  visited=set() /* creates a empty set                                           */
(4)  Queue=[] /* creates a empty Queue                                              */
(5)  seedNode=random.choice(G_s.nodes())
(6)  Queue.append(seedNode)
(7)  visited.add(seed)
(8)  while Queue is not empty do
(9)      if Queue is empty then
(10)         curr=[]
(11)         foreach node in G.nodes() do
(12)             if node not in visited then
(13)                 curr.append(node)
(14)             end
(15)         end
(16)         Queue.append(random.choice(curr))
(17)     end
(18)     u=Queue.pop()
(19)     G_s.add_node(u)
(20)     foreach v in G.neighbours(u) do
(21)         if v not in visited then
(22)             Queue.append(v)
(23)             G_s.add_edge(u,v)
(24)             visited.add(v)
(25)             if G_s.numberOfNodes ≥ sampleNodeCount then
(26)                 break
(27)             end
(28)         end
(29)     end
(30) end
```

3.3.7 Tightly Induced Breadth First Search Sampling

Here BFS is followed by an Induction step. while doing BFS nodes are sampled until a budget is reached. Then in the Induction step all the edges present in graph G is added between every sampled node. BFS has been used by researchers in the past but BFS along with the tightly induction step is not studied before.

Algorithm 7: Tightly Induced Breadth First Search Sampling

Result: $G_s(V_s, E_s)$

```
(1)  sampleNodeCount = G.numberOfNodes * δ
(2)  G_s=Graph()/* creates an empty undirected unweighted graph */
(3)  visited=set() /* creates a empty set */
(4)  Queue=[] /* creates a empty Queue */
(5)  seedNode=random.choice(G_s.nodes())
(6)  Queue.append(seedNode)
(7)  visited.add(seed)
(8)  while Queue is not empty do
(9)      if Queue is empty then
(10)         curr=[]
(11)         foreach node in G.nodes() do
(12)             if node not in visited then
(13)                 curr.append(node)
(14)             end
(15)         end
(16)         Queue.append(random.choice(curr))
(17)     end
(18)     u=Queue.pop()
(19)     G_s.add_node(u)
(20)     foreach v in G.neighbours(u) do
(21)         if v not in visited then
(22)             Queue.append(v)
(23)             G_s.add_edge(u,v)
(24)             visited.add(v)
(25)             if G_s.numberOfNodes ≥ sampleNodeCount then
(26)                 break
(27)             end
(28)         end
(29)     end
(30) end
     /* induction step */
(31) sampleNodes=G_s.nodes()
(32) foreach node in sampleNodes do
(33)     foreach neighbour in G[node].neighbours do
(34)         if neighbour in sampleNodes then
(35)             G_s.addEdge(node,neighbour)
(36)         end
(37)     end
(38) end
```

3.4 Experimental Evaluation

3.4.1 Datasets

In this work we have considered 6 large graphs namely hepPh, condMat, emailEnron, tree, tree2, tree3. hepPh is directed graph, which is a representation of paper citation network of Arxiv High Energy physics category. condMat is a directed graph which represents the collaboration network of Arxiv Codensed Matter category. emailEnron is a directed graph which represents the Enron email network. tree1, tree2, tree3 are trees which are created using pythons networkx module. This module can be used to produce random trees of specific size. Table 3.1 specifies the size of graph in details.

Dataset	Nodes	edges
hepPh	34546	421578
condMat	23133	186936
emailEnron	36692	367662
tree1	10000	9999
tree2	20000	19999
tree3	30000	29999

Table 3.1: Datasets

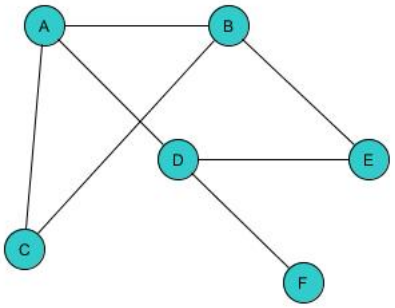
3.4.2 Properties

The properties that are being considered in this work as follows:

1. Degree Distribution
2. Clustering Coefficient
3. Largest Connected Component Size
4. Average Shortest Path Length

3.4.2.1 Degree Distribution

The degree of a node in a graph is the number of adjacent nodes to which it is connected. The probability distribution of these degrees over the whole graph is Degree Distribution. The degree distribution P(k) of a network is then defined to be the fraction of nodes in the network with degree k. Thus if there are n nodes in total in a network and n_k of them have degree k, we have $P(k) = n_k/n$. Refer to Fig 3.1.



Degree Distribution

Degree	Number of nodes	proportion
1	1	1/6 = 0.1667
2	2	2/6 = 0.3333
3	3	3/6 = 0.5000

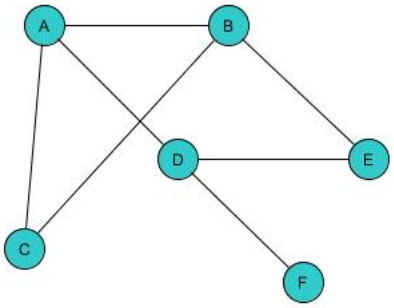
Distribution(d)= number of nodes having degree d / total number of nodes

Figure 3.1: Degree Distribution

3.4.2.2 Clustering Coeffcient

The clustering coefficient of a vertex is a measure of how close its neighbors are to being a clique. A graph G = (V, E) where V is the set of vertices and E is the set of edges. And edge e_{ij} is a connection between vertex v_i and v_j. The neighbor N_i for a vertex v_i represents the immediate neighbor of vertex v_i.
$N_i = \{v_s : e_{st} \in EVe_{ji} \in E \}$
The clustering coefficient v_i is C_i.
$C_i = 2\frac{|\{e_{jk}:v_j,v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i-1)}$
where k_i is the number of neighbors of vertex v_i.

Figure 3.2 represents the Clustering Coefficient of nodes and Fig. 3.3 represents the Clustering Coefficient Distribution.



Node Label	Clustering coefficient
A	1/3 = 0.3333
B	1/3 = 0.3333
C	1/1 = 1.0000
D	0/3 = 0.0000
E	0/1 = 0.0000
F	1/1 = 1.0000

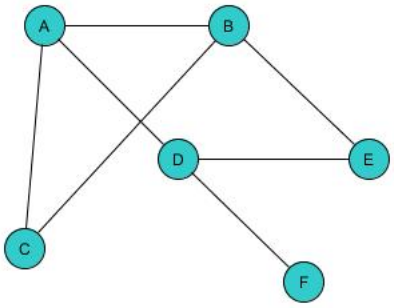
Figure 3.2: Clustering Coeffcient of nodes

3.4.2.3 Largest Connected Component Size

A connected component is a subgraph in which every vertex is connected to every other vertex and which is not connected to any other vertex in the supergraph. The component size is the number of vertices in the component. In our work we have considered the component having largest size out of all the components in the given graph. It can be used to assess the reachability of one node from another node. Lets assume two nodes A and B which are connected in the original graph. If the LCC of sample graph is very less compared to the expected size, then it is hardly possible that A is reachable from B or vice versa. On the other hand, if the LCC is same as the expected LCC of the sample then it is highly probable that both the nodes are connected. If we give more priority to reachability of nodes, then we should consider an algorithm which creates a sample whose LCC is same as the expected LCC of the sample. Refer to Fig 3.5. weakly connected component distribution has been studied in the past but none of the researchers have considered the largest connected component for comparing sampling algorithms.

3.4.2.4 Average Shortest Path Length

The average shortest path length is one of the measure of networks structure. Examples include, the average number of hops to transport data packets from source to destination, the average number of clicks which will lead you from one website to another. ASPL is in use since a long time to compare sampling algorithms. The ASPL along with LCC can be used to compare samples. If the LCC of two samples are almost same then the algorithm which exhibits lower ASPL is better than the other algorithm. It can be defined as follows. Consider an unweighted graph directed graph G = (V, E), where V is the set of vertices and E is the set of edges. Let



Node Label	Clustering coefficient
A	1/3 = 0.3333
B	1/3 = 0.3333
C	1/1 = 1.0000
D	0/3 = 0.0000
E	0/1 = 0.0000
F	1/1 = 1.0000

Distribution

Range	Numebr of nodes	Proportion
0.00-0.10	2	2/6 = 0.3333
0.10-0.20	0	0/6 = 0.0000
0.20-0.30	0	0/6 = 0.0000
0.30-0.40	2	2/6 = 0.3333
0.40-0.50	0	0/6 = 0.0000
0.50-0.60	0	0/6 = 0.0000
0.60-0.70	0	0/6 = 0.0000
0.70-0.80	0	0/6 = 0.0000
0.80-0.90	0	0/6 = 0.0000
0.90-0.10	2	2/6 = 0.3333

Figure 3.3: Clustering Coefficient Distribution

distance (v_i , v_j) where v_i , $v_j \in V$ denotes the shortest distance between the vertex v_i and v_j. Assuming distance (v_i , v_j) = 0 if there is no path from v_i to v_j. Then the average path length l_G is
$l_G = \frac{1}{n(n-1)} \sum_{i \neq j} distance(v_i, v_j)$ Refer to Fig 3.4.

3.5 Evaluation Measures

In this work the properties we are considering are clustering coefficient, degree distribution, average shortest path length and largest connected component. To measure the efficiency of sampling algorithms implemented we compare the Probability Density Function(PDF) as well as Cumulative Density Function(CDF) of each of these four properties. Apart from visually comparing the difference in properties, we also compare it quantitatively by two statistics namely Kolmogorov-Smirnov (KS) and Kullback-Leibler (KL).

3.5.1 Kolmogorov-Smirnov (KS)

KS difference is the maximum vertical distance between two given distributions. Let D_1 and D_2 represent two CDFs and i represents the range of random variable.
$KS(D_1, D_2) = \max_i \mid D_1(i) - D_2(i) \mid$ Figure 3.6 represents 2 distributions Distribution 1 and Distribution 2. KS difference between these two distributions is 0.2, as it is the maximum difference.

3.5.2 Kullback-Leibler (KL)

KL measures the average number of extra bits required to represent samples from the original distribution when using the sampled distribution. Let P and Q are two distributions where P represents the distribution of

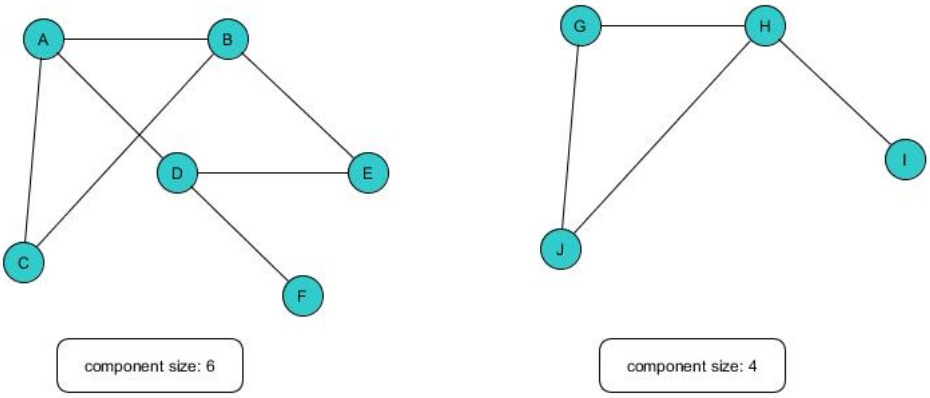


Figure 3.4: Connected Component

original graph and Q represents the distribution of sample graph.
KL (P || Q) = $\sum_{i=1}^{N} P(x_i) \log_2 \frac{P(x_i)}{Q(x_i)}$
KL (P || Q) is not defined for distributions that have some values with 0 probabilty. To resolve this issue, we use Skew-Divergence which smooths the PDFS before computing the KL divergence.
SD(P_1 ,P_2 , α) = KL[α *P_1 + (1-α) *P_2 || α *P_2 + (1-α) *P_1]
Figure 3.7 represents the smoothening step. After smoothening the two distribution we can use the KL formula to evaluate the KL divergence. Please note $\alpha = 0.99$ is used throughout this work.

3.6 Results

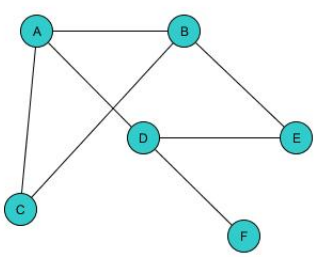
We obtain a sample between 5% to 50%($\alpha = 0.05$ to 0.50) of the original graph. This sampling range can clearly emphasize the similarity in properties between the sample and original graph as we increase the sampling factor. For every sampling factor we create 10 samples.

Firstly, we will compare the implemented algorithms visually from their cumulative distribution(CDF) for degree distribution and clustering coefficient distribution. Then the average SD distance and KS distance is computed for all the 10 different samples of every factor and we do this for every dataset.

Degree Distribution Figure 3.8 a, 3.8 b and 3.8 c represents the cumulative distribution function(CDF) for graphs. From the figure it is clear that Node Sampling(NS) and Forest Fire Sampling(FFS) misjudge the degree of nodes which results in the sample having large number of zero degree nodes in all the samples. FFS is slightly better than NS. Across the 3 figures TIES, KARGER, TIBFS estimates the degree close to the actual graph. It is expected that TIES, KARGER, TIBFS will sample the high degree nodes. In these three algorithms we randomly sample an edge, the probability of choosing a high degree node is high. The probability of a node being picked is directly proportional to the number of edges it is having, which is the degree of the node. This very fact helps in preserving the degree distribution more accurately.

Figure 3.9 a, 3.9 b and 3.9 c represents the cumulative distribution function(CDF) for trees. In trees, the difference in performance of TIES and ES is very less compared to the difference we encountered in graphs. TIES couldn't get much advantage in the induction step because it can only add few edges between already sampled nodes. On the other hand, KARGER, BFS, TIBFS gives a more accurate sample of the original graph. These three algorithms are traversal based and do not divide the tree into different components which results in more accurate estimation of degree distribution.

Clustering Coefficient Distribution Figure 3.10 a, 3.10 b and 3.10 c represents the cumulative distribution function(CDF) for graphs. BFS and ES underestimates the distribution. In ES an edge is sampled uniform at random, it doesn't depend on the neighbors of already sampled nodes. In BFS we add the seed node and then its neighbors. we only add the edge between node and its parent. Remaining edges between nodes are



from A

TO	shortest Path Length
B	1
C	1
D	1
E	2
F	2

from B

TO	shortest Path Length
A	1
C	1
D	2
E	1
F	3

from C

TO	shortest Path Length
A	1
B	1
D	2
E	2
F	3

from D

TO	shortest Path Length
A	1
B	2
C	2
E	1
F	1

from E

TO	shortest Path Length
A	2
B	1
C	2
D	1
F	2

from F

TO	shortest Path Length
A	2
B	3
C	3
D	1
E	2

Average Shortest path length=(7+8+9+7+8+11)/(6*(6-1))=1.6667

Figure 3.5: Average Shortest Path Length

lost and hence we lose the clustering score of sampled nodes. The induction step helps in getting back the clustering score of a sample node as it increases the number of edges between neighbors. Again TIES, TIBFS and KARGER are performing better than other algorithms.

Figure 3.11 a, 3.11 b and 3.11 c represents the cumulative distribution function(CDF) for trees. The clustering coefficient of every node in a tree is 0. There are no edges between neighbors in a tree because a single edge between two nodes will form a cycle. So the curve is same for all the algorithms. As it is a CDF and the proportion of nodes having 0.00 clustering coefficient is 1 so the whole proportion cumulatively evaluates to 1 after that.

Largest Connected Component Size Figure 3.12 a, 3.12 b and 3.12 c represents the largest connected component(LCC) size of graphs. The red colored horizontal bar is the expected size of the LCC. BFS, KARGER, TIBFS overestimates the size of LCC size. In all these three algorithms the component size will be dictated more or less by the seed node. If the seed node belongs to the largest component, then it will sample all the nodes from this component. The probability of choosing a seed node from the LCC is high as compared to nodes in other components. TIES performs better than other algorithms for this property. The number of edges is expected to be high in the LCC, so more than often it will end of choosing a node from the LCC and it samples from the other components also. So TIES is able to remove this over counting from the LCC alone.

Figure 3.13 a, 3.13 b and 3.13 c represents the largest connected component(LCC) size of trees. Here obviously KARGER, TIBFS and even BFS will work well because there is only component. So each one of it sample all the nodes from this component and hence completely preserve the LCC size. TIES don't work well because in trees TIES loose the advantage of induction step and TIES is not a traversal based algorithm, so it divides the component to many other components.

Average Shortest Path Length Figure 3.14 a, 3.14 b and 3.14 c represents the average shortest path length(ASPL) of graphs. The average shortest path length of a tree will be higher compared to a general graph. The ES algorithm shows a high ASPL than other algorithms. This algorithm can be thought of forming a tree like structure. The ASPL of hepPh is less as it is denser than the condMat and emailEnron. So it is possible that

KS Test			
	Distribution 1	Distribution 2	Difference
	0.2	0.3	0.1
	0.3	0.1	0.2
	0.1	0.2	0.1
	0.1	0.2	0.1
	0.3	0.2	0.1

Figure 3.6: KS

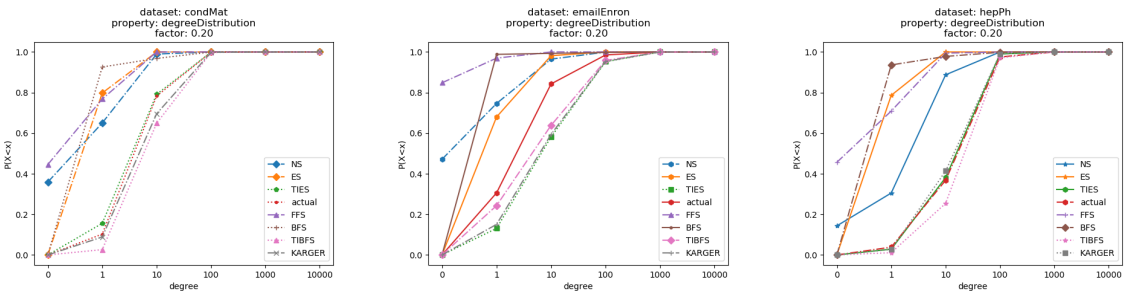
KL Test				KL Test	
	Distribution 1	Distribution 2		Distribution 1	Distribution 2
	0.2	0.3		0.201	0.299
	0.3	0.1	After Smoothening	0.298	0.102
	0.1	0.2		0.101	0.199
	0.0	0.2		0.002	0.198
	0.3	0.2		0.299	0.201

Figure 3.7: KL

ES overcomes its tree like structure in a dense environment. KARGER, TIBFS, TIES has almost same LCC so they are comparable based on ASPL. For condMat and emailEnron TIES has largest ASPL but for hepPh sample it goes down as hepPh is denser than the other two graphs. Denser a graph lesser will be the ASPL. TIES performs better in case of denser graphs and KARGER, TIBFS performs better in case of sparse graph.

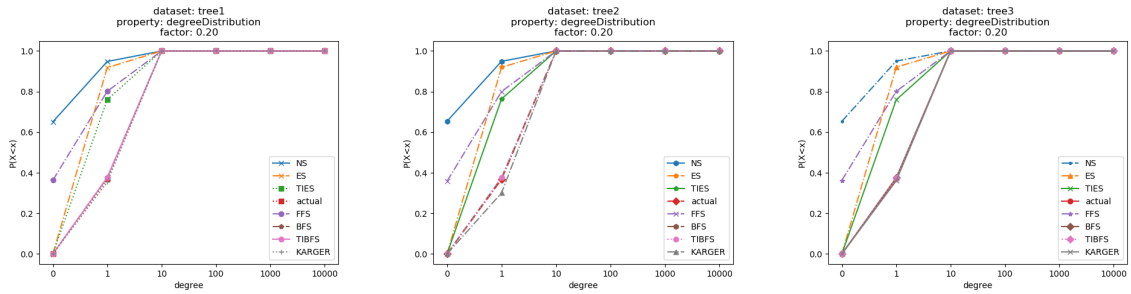
Figure 3.15 a, 3.15 b and 3.15 c represents the average shortest path length(ASPL) of trees. ES, TIES, NS, BFS divides the tree into several small sized components as it is also evident from the Fig.3.13 a, b, c. A smaller component will have lesser ASPL. BFS, KARGER, TIBFS do not divide the tree and hence end up with having higher ASPL. The LCC of trees sampled by BFS, TIBFS, KARGER are same but KARGER has largest ASPL, so BFS and TIBFS is preferable over KARGER in case of trees.

Summary Overall TIBFS and KARGER is performing better than other algorithms. In case of general graphs TIES works well and TIBFS and KARGER is slightly away from the actual curve. For trees TIBFS and KARGER outperforms if we consider all the properties. TIES is an edge based selection algorithm which



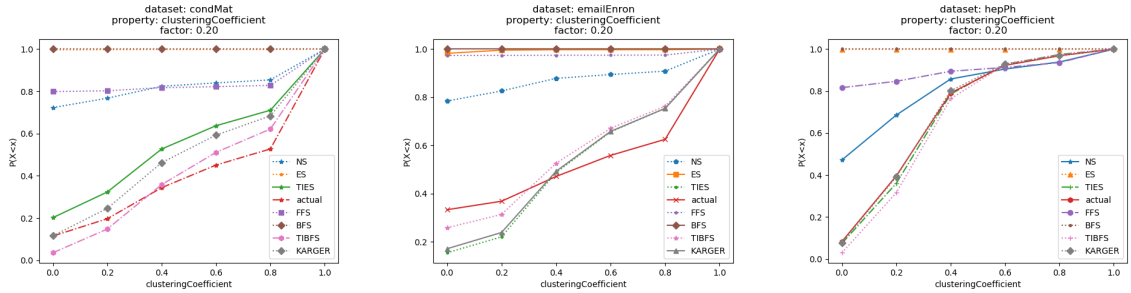
(a) condMat (b) emailEnron (c) hepPh

Figure 3.8: Degree Distribution for Graphs



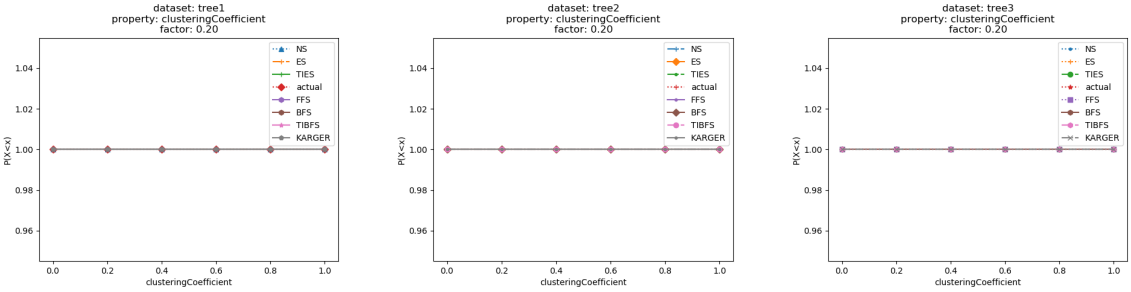
(a) tree1 (b) tree2 (c) tree3

Figure 3.9: Degree Distribution for Trees



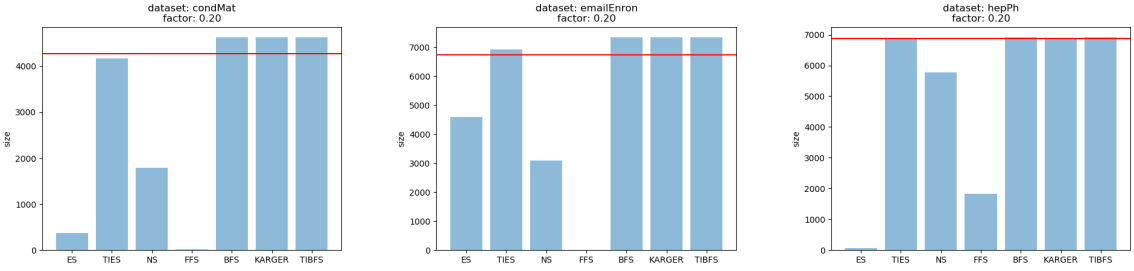
(a) condMat (b) emailEnron (c) hepPh

Figure 3.10: Clustering Coefficient Distribution for Graphs



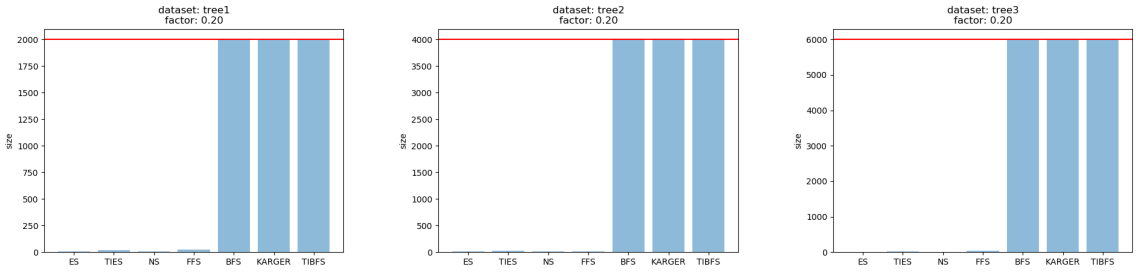
(a) tree1 (b) tree2 (c) tree3

Figure 3.11: Clustering Coefficient Distribution for Trees



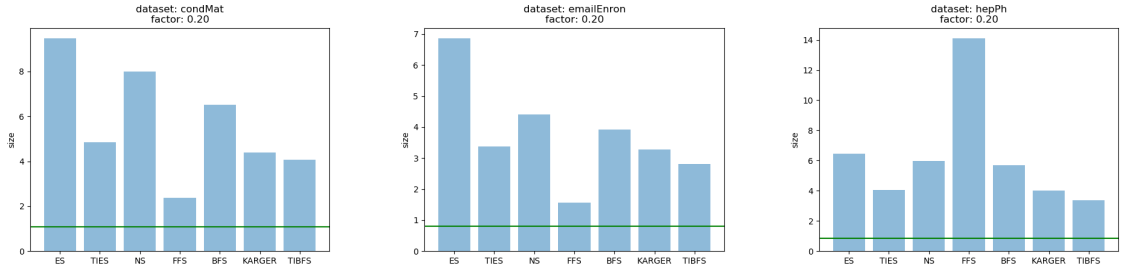
(a) condMat (b) emailEnron (c) hepPh

Figure 3.12: Largest Connected Component Size of Graphs



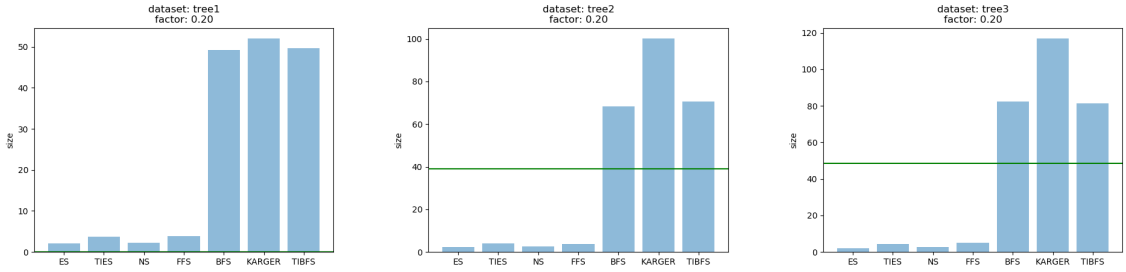
(a) tree1 (b) tree2 (c) tree3

Figure 3.13: Largest Connected Component Size of Trees



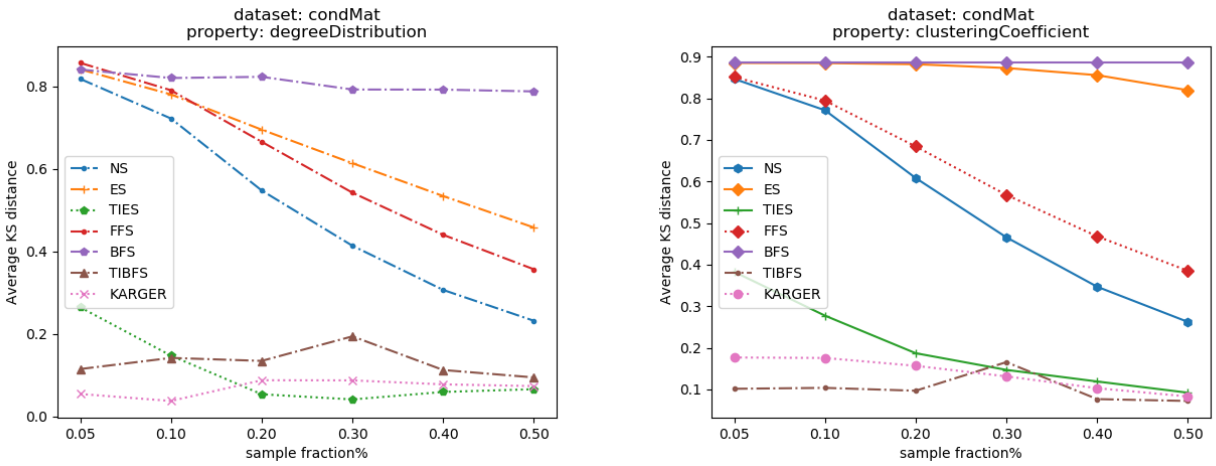
(a) condMat (b) emailEnron (c) hepPh

Figure 3.14: Average Shortest Path Length of Graphs



(a) tree1 (b) tree2 (c) tree3

Figure 3.15: Average Shortest Path Length of Trees



(a) Degree Distribution

(b) Clustering Coefficient

Figure 3.16: KS statistic for condMat

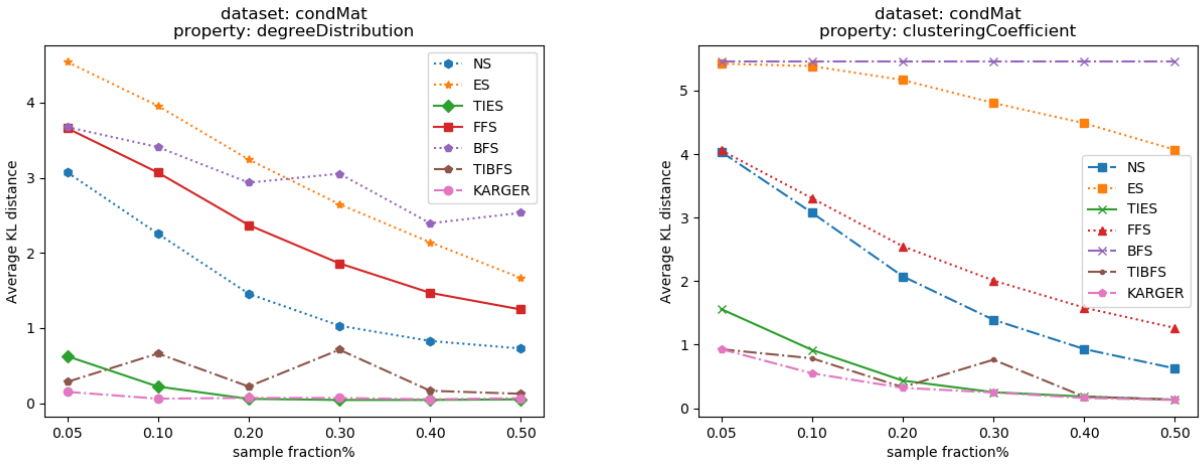
samples high degree nodes and in the induction step it includes all the edges between sampled nodes but it is unable to take the advantage of induction step in case of trees. Hence its performance decreases in case of trees. The traversal based algorithm (TIBFS, BFS) overestimates the size of LCC as it samples all the nodes from the LCC of the original graph. But TIES escapes from this bias by adding edges from smaller components. KARGER and TIBFS works well for ASPL as well. It keeps the ASPL lesser which is better than having a high ASPL. For trees KARGER and TIBFS didn't work well because it doesn't divide the tree into components unlike other algorithms.

KS-statistic For every factor we have already created 10 samples. we find the average KS distance for this 10 samples and we do the same for all datasets. Figure 3.16, 3.18, 3.20, 3.22, 3.24, 3.26 represents the average KS distance for every sampling fraction. we have considered degree distribution and clustering coefficient distribution for evaluation. In graphs TIES, KARGER, TIBFS is performing well than other algorithms. Overall, the sampling algorithms which include induction step as a part of its algorithm works well for both the properties. Coming to trees TIES lose its advantage of induction step and do not perform well. The performance of TIBFS is better than KARGER. The traversal based algorithm performs well in case of trees for degree distribution. Even BFS performs well in case of trees. The performance of most algorithms increase as we increase the sampling size.

Skew Divergence KS statistic evaluates the maximum distance between two given distribution but the skew divergence presented in Fig. 3.17, 3.19, 3.21, 3.23, 3.27 captures the divergence between two distributions across all factors. We can observe that for graphs TIES, KARGER, TIBFS exhibits lesser skew than any other algorithm. In case of trees, again traversal based algorithm like TIBFS, BFS, KARGER has lesser skew than other algorithms.

3.7 Conclusion

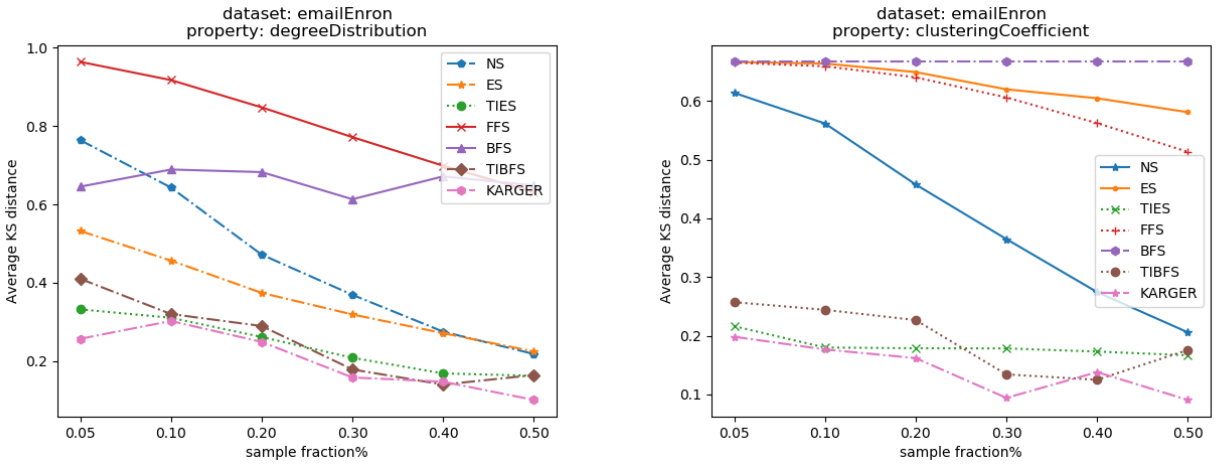
In this work we have presented the performance of several sampling algorithms on graphs and trees. we have found that TIES, KARGER, TIBFS works well on graphs and TIES loses its advantage of induction step in case of trees and its performance decreases. Please note that KARGER algorithm is computationally expensive than other algorithms. If a graph is dense then TIBFS is preferable over KARGER. Previous works reflects that TIES produces a sample which closely follow the distribution of the original graph. We showed that TIES is not following the distribution of original graph in case of trees. we present three traversal based algorithms i.e. TIBFS, KARGER, BFS which performs better than other algorithms.



(a) Degree Distribution

(b) Clustering Coefficient

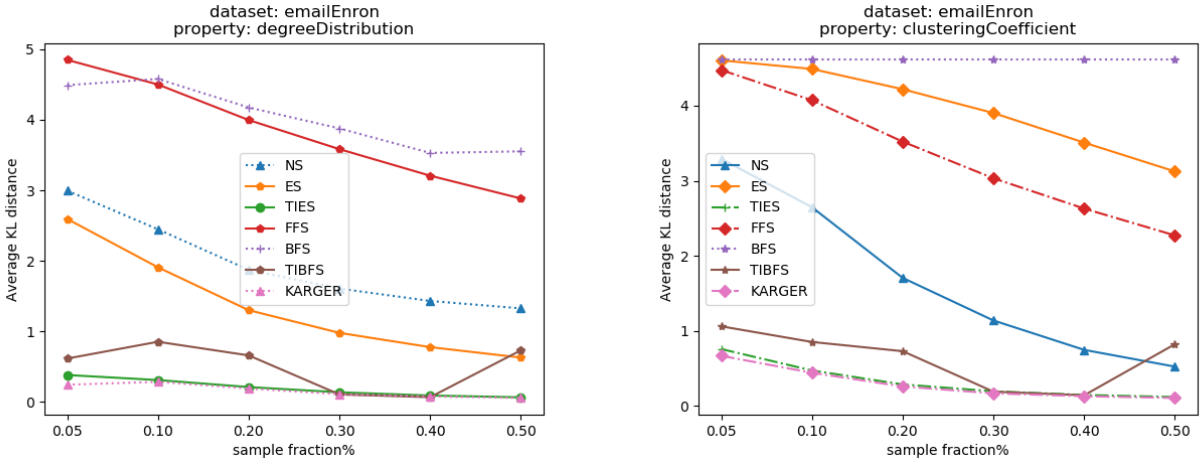
Figure 3.17: Skew Divergence for condMat



(a) Degree Distribution

(b) Clustering Coefficient

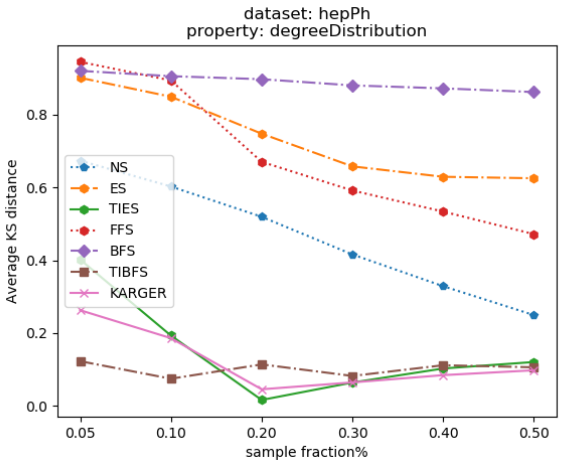
Figure 3.18: KS statistic for emailEnron



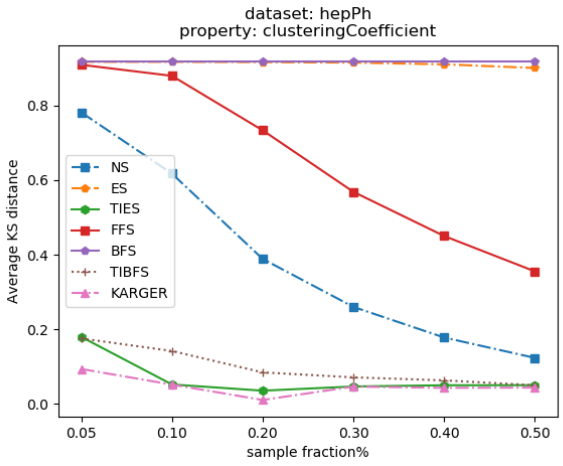
(a) Degree Distribution

(b) Clustering Coefficient

Figure 3.19: Skew Divergence for emailEnron

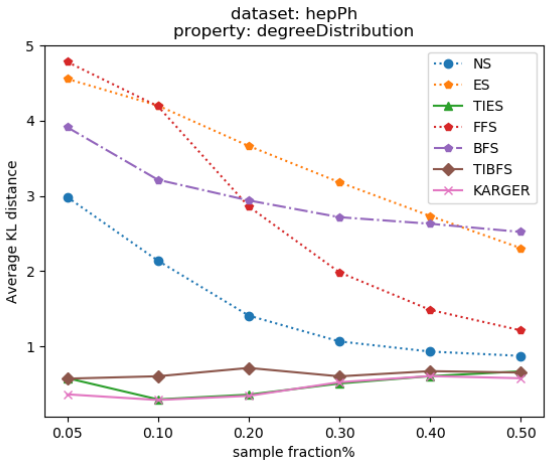


(a) Degree Distribution

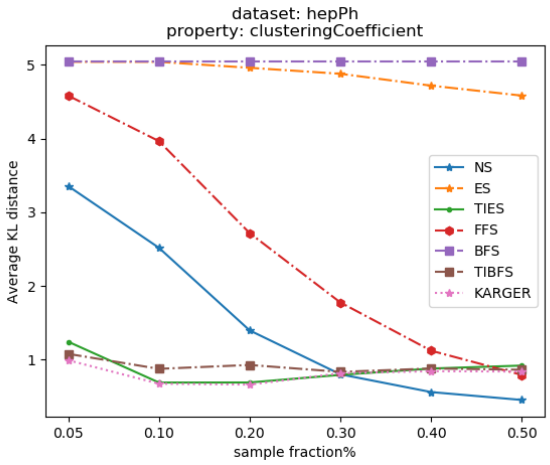


(b) Clustering Coefficient

Figure 3.20: KS statistic for hepPh

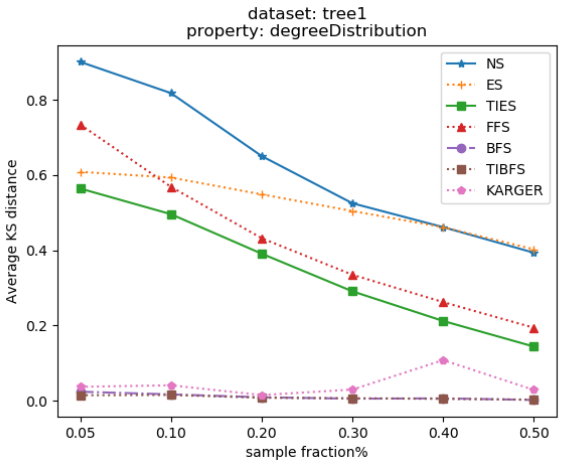


(a) Degree Distribution

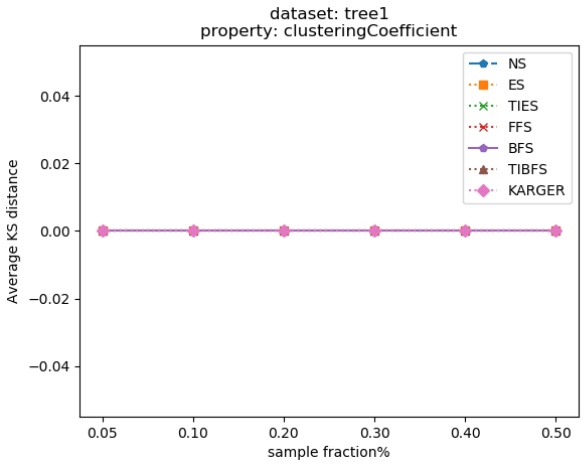


(b) Clustering Coefficient

Figure 3.21: Skew Divergence for hepPh

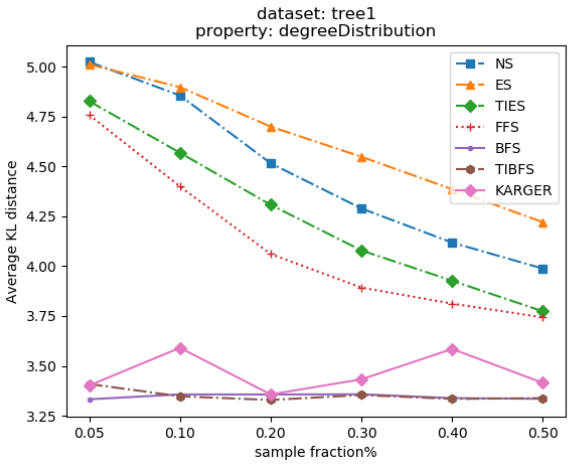


(a) Degree Distribution

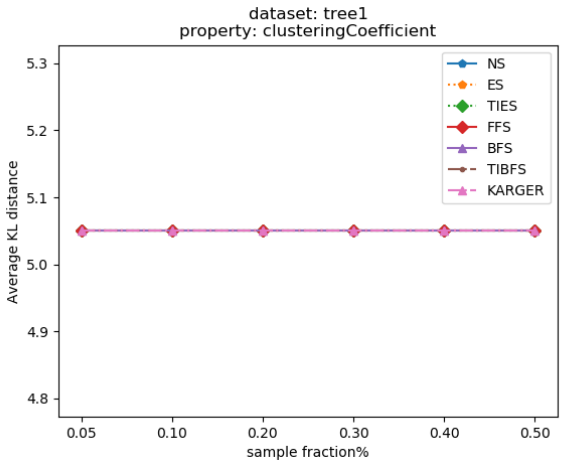


(b) Clustering Coefficient

Figure 3.22: KS statistic for tree1

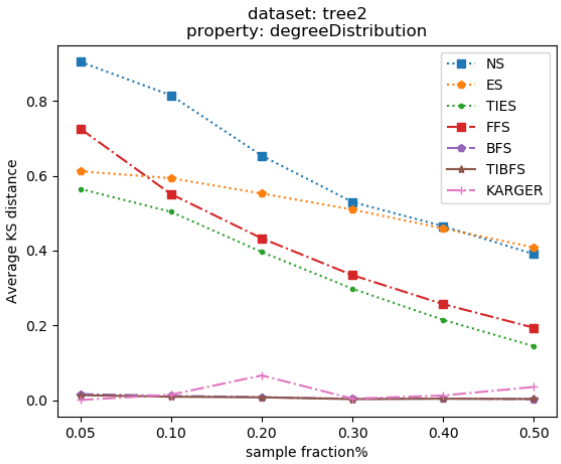


(a) Degree Distribution

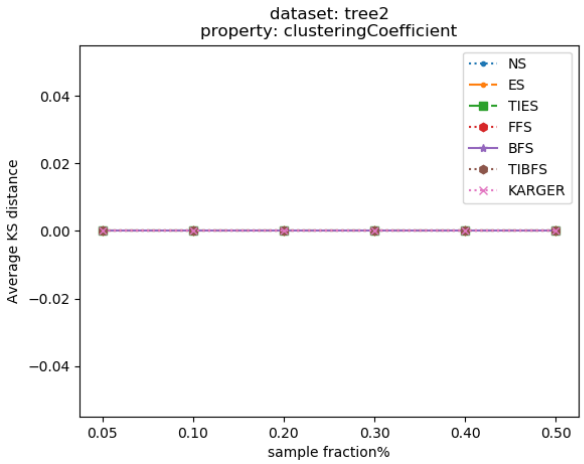


(b) Clustering Coefficient

Figure 3.23: Skew Divergence for tree1

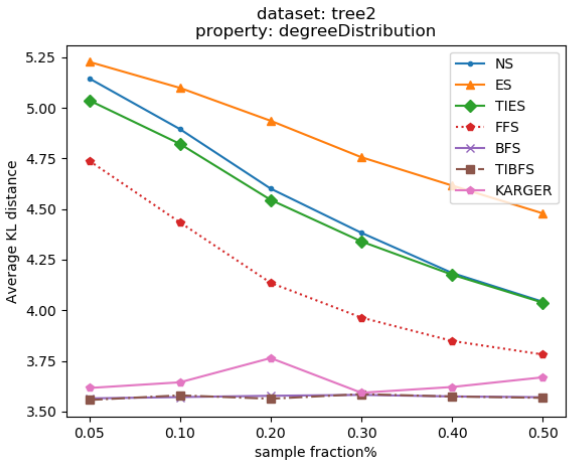


(a) Degree Distribution

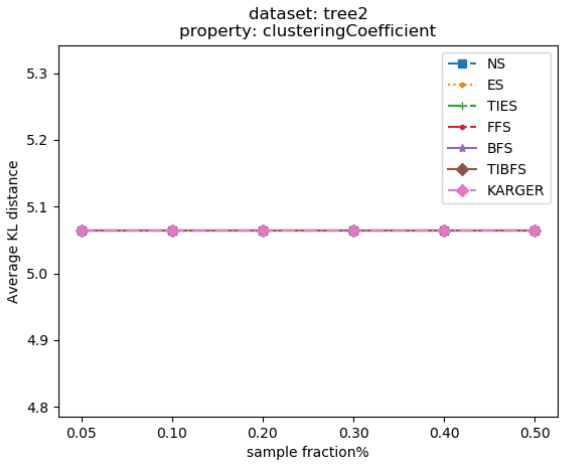


(b) Clustering Coefficient

Figure 3.24: KS statistic for tree2

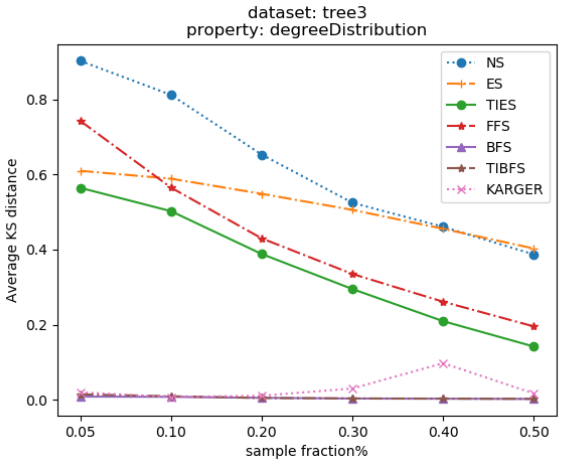


(a) Degree Distribution

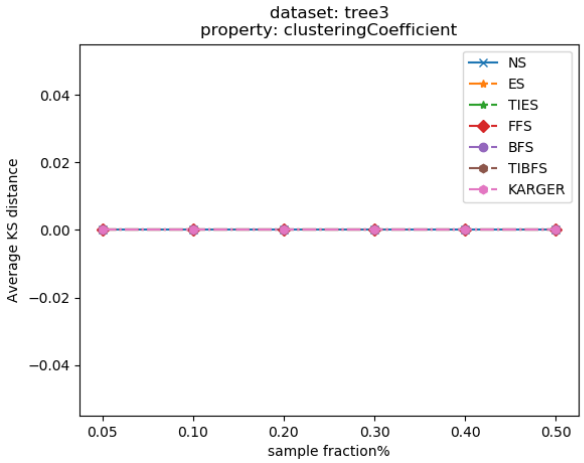


(b) Clustering Coefficient

Figure 3.25: Skew Divergence for tree2

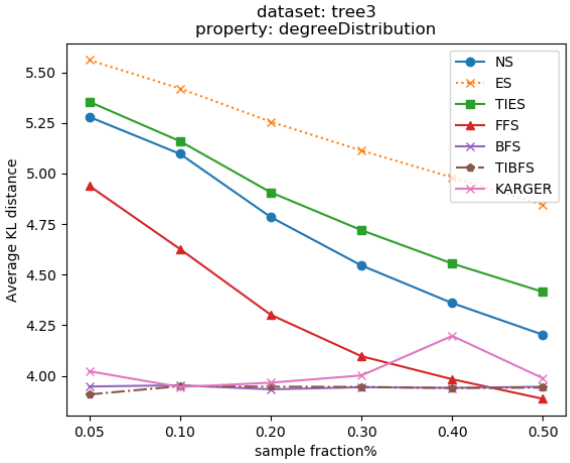


(a) Degree Distribution

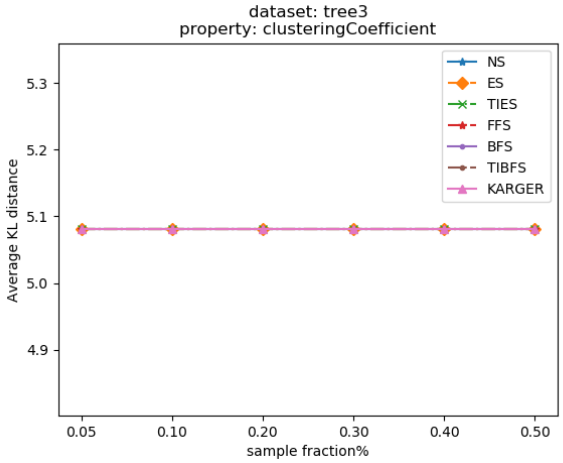


(b) Clustering Coefficient

Figure 3.26: KS statistic for tree3



(a) Degree Distribution



(b) Clustering Coefficient

Figure 3.27: Skew Divergence for tree3

In future more analysis of the algorithms can be done on special graphs like a complete graph, cycle graph, bipartite graph and so on. Computational time can be considered as a parameter for evaluation purpose.

References

[1] R. A. Hearn and E. D. Demaine. Constraint Logic: A Uniform Framework for Modeling Computation as Games. *IEEE* 1–14.

[2] E. D. D. Robert A. Hearn. Games, Puzzles, and Computation. 1st edition. A K Peters, Ltd, 2009.

[3] T. K. Shigeki Iwata. The Othello game on an n x n board is PSPACE-complete. *Elsevier Science* 123, (1994) 329–340.

[4] T. K. Shigeki Iwata. Generalized Hi-Q is NP-complete. *THE TRANSACTIONS OF THE IEICE* 73, (1990) 270–273.

[5] D. P. H. Robert A. Beeler. Peg solitaire on the windmill and the double star graphs. *AUSTRALASIAN JOURNAL OF COMBINATORICS* 52, (2012) 127–134.

[6] P. Hu and W. C. Lau. A Survey and Taxonomy of Graph Sampling .

[7] R. R. K. Nesreen Ahmed, Jennifer Neville. Network Sampling via Edge-based Node Selection with Graph Induction. *Department of Computer Science Technical Reports* 1–10.